Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN COMPUTACIÓN

WorldGenerator: un sistema de generación procedural de grandes entornos naturales 3D basado en la combinación de mapas de ruido y muestreos de Poisson.

Estudante: Andrés Fariñas Riveiro
Dirección: Enrique Fernández Blanco
Daniel Rivero Cebrián

A Coruña, xuño de 2021.

A todo aquel que alguna vez estuvo, está o estará en mi vida.

Agradecimientos

A Dani y Quique, que sin su apoyo y consejos este trabajo no habría sido posible.
A mis amigos, por permanecer ahí pase lo que pase.
A mi padre, por inculcarme que se debe pensar antes de actuar.
A mi madre, por apoyarme siempre independientemente de lo que hiciese con mi vida.
A Lidia, por aguantarme todos estos años y ser un pozo inagotable de felicidad.
A todo el que lea este trabajo, que se lleva una parte de mí en su interior.

Resumen

El mundo del modelado de objetos 3D se ve cada vez más catapultado por el aumento de inversión en las industrias de videojuegos, simulación y diseño. Concretamente en simulación de conducción o pilotaje de vehículos y en desarrollo de videojuegos se usan cantidades elevadas de tiempo y dinero en la etapa de modelado de los entornos naturales.

El presente proyecto desarrollará una herramienta que genere de forma procedural los entornos 3D. El sistema estará enfocado en el desarrollo de grandes extensiones en las cuales se combinen diferentes biomas con transiciones naturales. Los elementos decorativos se colocarán conforme a distribuciones pseudoaleatorias, con naturalidad pero respetando las distancias. Toda la generación y colocación estará condicionada por una semilla, que garantice la reproducibilidad, es decir, que todo fragmento de terreno o decorado se genere exactamente igual independientemente de cuántas veces se cree o destruya.

Abstract

The world of 3D modeling is blooming by the increasing investment in the video game, simulation and design industries. More specifically, in vehicle driving simulation and video game development, high amounts of time and money are spent on modelling natural environments. This project aims to develop a tool that procedurally generates those 3D environments. The resulting system will be focused on the development of large expanses in which different biomes are combined with natural transitions. Also, decorative elements will be placed according to pseudo-random distributions, in order to mimick the nature but respecting distances. The whole generation and placement shall be conditioned by a random seed, which guarantees the reproducibility, i.e. that every fragment of terrain or decoration is generated in exactly the same way no matter how many times it is created or destroyed.

Palabras clave:

- Generación procedural
- Muestreos de Poisson
- Mapas de ruido
- Generación de terreno
- Generación de ecosistemas
- Unity3D
- Simulador

Keywords:

- Procedural generation
- Poisson samplings
- Noise maps
- Terrain generation
- Ecosystem generation
- Unity3D
- Simulator

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

1.1 Motivación

El mundo del modelado de objetos 3D se ve cada vez más catapultado por el aumento de inversión en las industrias como los videojuegos, la simulación y el diseño.

Concretamente, en simulación de vehículos y en desarrollo de videojuegos, se usan cantidades elevadas de tiempo y dinero en la etapa de modelado de los entornos naturales.

Estos entornos constan de un terreno y una serie de elementos decorativos. El terreno está subdividido en biomas, que son el conjunto de factores climáticos y geológicos que caracterizan una región, y estos determinan los tipos de elementos decorativos. Los terrenos han de ser modelados a mano, usando herramientas de diseño 3D, empleando en cada metro cuadrado minutos o incluso horas. Los elementos decorativos se han de colocar con precisión, respetando la gravedad, las distancias, el realismo, etc. En cada kilómetro cuadrado puede haber de cientos a decenas de miles de elementos decorativos, cada uno de ellos colocado a mano. Por tanto, un entorno de unos pocos kilómetros cuadrados puede tener un coste astronómico, de varios meses de trabajo para un equipo de múltiples diseñadores 3D.

El alto coste, junto al uso limitado que se le pueden dar a estos escenarios, dificulta el acceso a este sector. Por ejemplo, los entornos de un videojuego no se emplean en ningún otro, es decir, tienen un solo uso. Una desarrolladora pequeña difícilmente podría plantearse la creación de este tipo de sistemas.

Este problema se soluciona con herramientas de generación automatizada de entornos 3D, que transforman los meses de trabajo en minutos, reduciendo drásticamente los recursos empleados. Pero estas herramientas generan otro problema, las empresas que las desarrollan no las comercializan, ya que darían una ventaja significativa a la competencia.

El presente proyecto pretende acortar el tiempo y dinero empleados en el modelado del entorno, para lo cual se desarrollará una herramienta que genere de forma procedural los entornos 3D. El sistema estará enfocado en el desarrollo de grandes extensiones en las cuales se

combinen diferentes biomas con transiciones naturales. El terreno y los elementos decorativos se generarán y destruirán de forma dinámica, de acuerdo a la distancia que se encuentren de la cámara que renderiza el mundo 3D en pantalla. Los elementos decorativos se colocarán conforme a distribuciones pseudoaleatorias, con naturalidad pero respetando las distancias. Toda la generación y colocación estará condicionada por una semilla, que garantice la reproducibilidad, es decir, que todo fragmento de terreno o decorado se genere exactamente igual independientemente de cuántas veces se cree o destruya mientras la semilla permanezca igual.

1.2 Objetivos

El objetivo principal del proyecto es la creación de una herramienta para el motor gráfico Unity3D que, basándose en unos parámetros introducidos por el usuario, genere un entorno natural. Dicho entorno estará compuesto, en primer lugar, de un terreno con distintos biomas y, en segundo lugar, de los decorados propios de los ecosistemas que componen cada bioma, para conseguir un entorno natural y realista. Dicho entorno servirá de base para otros sistemas más complejos, como videojuegos o simuladores de vehículos terrestres y aéreos, ahorrando recursos que se podrán destinar a otras áreas de desarrollo.

El usuario decidirá qué biomas se generan en el terreno y a que altitudes, asignando un conjunto de texturas al suelo (arena, hierba, nieve, etc.) que se aplicaran de forma automática en la generación. También se podrán personalizar los ecosistemas proporcionando los modelos 3D propios, como árboles, rocas, plantas, etc. El usuario decidirá la altura mínima y máxima (nivel del mar y cumbres montañosas), el tamaño del terreno y la semilla que genere el entorno.

Para la consecución de este objetivo se plantean las siguientes tareas:

- **Generación del terreno:** Definición de algoritmo de generación del terreno, que adecúe la altura del mismo a un mapa de altura, dicho mapa estará compuesto de la suma de las salidas de múltiples funciones de ruido.
- **Generación de biomas:** Definición de algoritmo suplementario para la generación de biomas, que, en función de la altura del terreno, asigne las texturas al suelo y en base a la salida de una función de ruido adicional separe el bioma en diferentes ecosistemas.
- **Creación y destrucción de elementos:** Definición de un sistema que, genere los elementos próximos y destruya los lejanos de la cámara que está visualizando el mundo 3D.
- **Colocación de elementos:** Definición de un algoritmo para el correcto posicionamiento de elementos en el suelo, basado en el principio de los muestreos de Poisson para evitar que objetos sólidos se superpongan.

Al mismo tiempo, se tendrán en cuenta los siguientes objetivos no funcionales:

- **Eficiencia del algoritmo:** El algoritmo ha de ejecutarse rápidamente ya que se busca ahorrar tiempo de modelado y diseño de entorno. El objetivo base es generar 1 kilómetro cuadrado de terreno y decorados por segundo.
- **Rendimiento del entorno generado:** El entorno ha de ser eficiente y no consumir recursos en exceso, ya que es la base para otros sistemas más complejos.
- **Realismo del entorno:** El entorno ha de ser lo suficientemente realista para pasar por uno creado a mano, no puede haber saltos de altura, agujeros o discontinuidad de los biomas.
- **Facilidad de uso:** La herramienta debe de poder usarse haciendo solo unos clicks con el ratón e introduciendo valores simples.

1.3 Estructura de la memoria

Para facilitar la lectura del presente documento, se añade una lista de los capítulos que lo componen, así como una breve descripción de cada uno.

1. **Introducción:** El presente capítulo. Ofrece una visión general del proyecto.
2. **Estudio de mercado:** Se realiza un análisis de herramientas similares, las que hay en el mercado y las que están integradas en otros sistemas, para estudiar la viabilidad del proyecto.
3. **Fundamentos:** Se estudia qué tecnologías usar, las bases necesarias de las mismas y los principios matemáticos necesarios para el correcto desarrollo del proyecto.
4. **Metodología y planificación:** Se explica el uso y adaptación de la metodología seleccionada, así como la planificación temporal y de costes inicial.
5. **Desarrollo:** En este apartado se detalla el trabajo realizado, dividido en las iteraciones detalladas en el capítulo anterior.
6. **Conclusiones:** Se realiza un análisis de la herramienta desarrollada para ver si se han cumplido los objetivos.
7. **Futuros desarrollos:** Se realiza una lista de tareas para aumentar las capacidades de la herramienta en un futuro desarrollo.

Capítulo 2

Estudio de mercado

Antes de la realización del proyecto se realizó un estudio sobre las diferentes herramientas que existen en el mercado, analizando lo que ofrecen y los aspectos que no cubren, sus fortalezas y debilidades. La finalidad de este estudio es la comprobación de la viabilidad de la herramienta, que posea suficientes ventajas y pocas o ninguna desventaja con respecto a la competencia.

El método convencional de creación de un escenario se define como, generación manual y colocación manual. De esta manera, el usuario debe crear los elementos a mano para, posteriormente, colocarlos en el escenario siguiendo un diseño.

Las herramientas actuales de generación procedural que podemos ver en el mercado se definen como, generación automática y colocación manual. El sistema es capaz de generar trozos de terreno y los elementos decorativos pertinentes, pero no realizan la fase de diseño de escenario, donde una persona ha de colocar a mano estos elementos. Si bien se podría usar un único trozo de terreno como escenario, este sería muy pobre en detalle, ya que cada trozo se basa en una plantilla y tendríamos, por ejemplo, un terreno compuesto únicamente de una montaña. Estaría más enfocado a un escenario de pruebas o demostración de potencia de la herramienta, no a un escenario totalmente funcional y extenso. Estas herramientas están orientadas a desarrolladores de videojuegos ya experimentados, con muchas opciones de personalización, pero gran dificultad para personas ajenas a ese sector.

Aunque también existen herramientas que se basan en generación manual y colocación automática, no se plantean en este estudio de mercado, ya que el proyecto está enfocado en la generación automatizada. Este tipo de herramientas se usan mayoritariamente en generación de ciudades, donde el usuario crea a mano todos los elementos (edificios, terrenos adaptados, semáforos, paradas de autobús, etc.) y se colocan automáticamente siguiendo una serie de reglas.

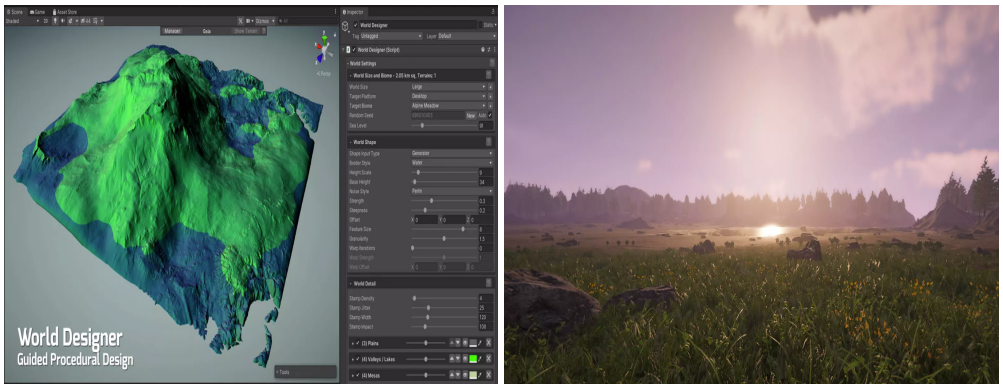
El tipo de herramienta que se pretende desarrollar se define como, generación automática y colocación automática. El sistema es capaz de, en base a unos parámetros, generar un terreno

y colocarlo de manera autónoma garantizando un nivel de realismo alto. Al mismo tiempo se colocan los elementos decorativos, correspondientes al tipo de terreno, de forma automática y respetando unas reglas.

2.1 Herramientas existentes en el mercado

Son las que mayor realismo generan, pero no solventan el principal objetivo de este proyecto. En estas herramientas, el diseñador sigue teniendo que colocar las piezas características del terreno a mano, una a una, la única parte que se suprime es el modelado del terreno, generado proceduralmente. La única ventaja que tienen estas herramientas es que la generación se realiza en edición y el usuario final no percibe este tiempo. A continuación se van a detallar algunas de estas herramientas:

- **Gaia Pro [1], de Procedural Worlds:** Implementada en Unity3D, esta herramienta parte de un conjunto amplio de imágenes como base para la generación del terreno, dichas imágenes son usadas como máscara aplicada a unos mapas de ruido. Esta generación se puede modificar con una serie de parámetros introducidos por el usuario y nos da como resultado un trozo de terreno, como se observa en la figura 2.1a, donde nos ofrece una isla con una montaña central. Estos trozos por sí mismos no son muy atractivos, por lo que habría que generar muchos y colocarlos a mano. Con muchos de estos trozos podemos componer una escena y generar un mundo. Lo que ofrece esta herramienta es un alto grado de personalización y calidad gráfica como se ve en la figura 2.1b. Lo que no ofrece es simplicidad, generación en segundos ni precio competitivo.

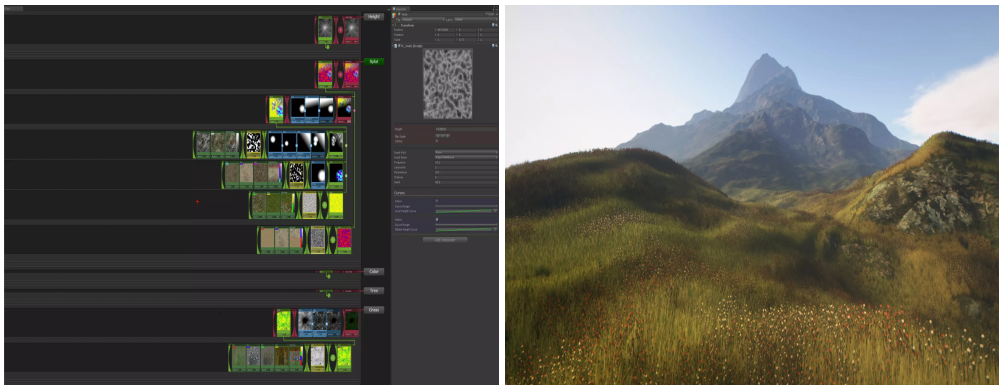


(a) Trozo de terreno generado por Gaia (b) Resultado final que ofrece Gaia

Figura 2.1: Gaia

- **Terrain Composer [2], de Nathaniel Doldersum:** Es una herramienta basada en nodos implementada en Unity3D. Presenta una interfaz como la que se muestra en la figura

2.2a, parte de una imagen como base para la generación. Cada imagen se puede conectar con otros nodos de ruido, máscaras, texturas, etc. Que aplicados, dan como resultado un fragmento de terreno. Para obtener un escenario complejo se han de combinar múltiples fragmentos. Ofrece un interfaz gráfico simple, un grado de personalización alto y buena calidad gráfica (véase figura 2.2b). Lo que no ofrece es rendimiento, tanto en generación como en ejecución, ni generación en segundos.



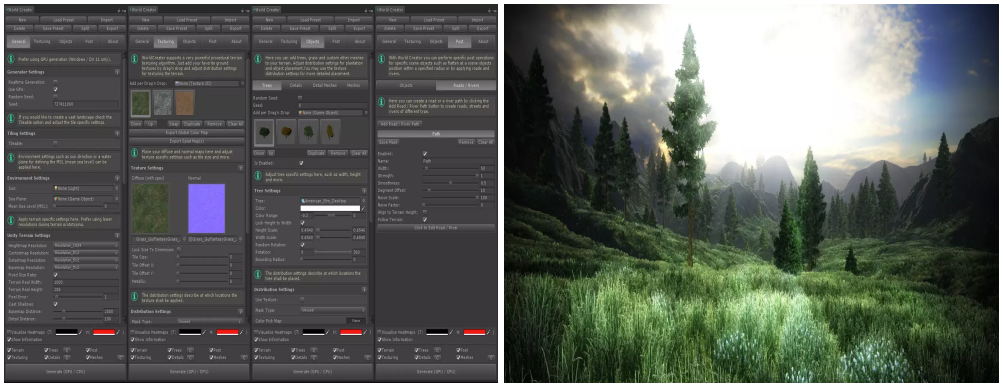
(a) Sistema de nodos de Terrain Composer (b) Resultado final de Terrain Composer

Figura 2.2: Terrain Composer

- **World Creator [3], de BiteTheBytes GmbH:** Es una herramienta basada en mapas de altura, implementada en Unity3D, que toma una semilla y genera un trozo de terreno. Dicho trozo es la base para la personalización posterior (ver figura 2.3b), que es muy amplia y detallada. Componiendo varios de estos trozos se crea manualmente el escenario. Como ventaja, está la inmensa personalización que se le puede dar al terreno, que alcanza un nivel de realismo muy alto como se ve en la figura 2.3a. Como desventajas, presenta una colocación manual de los elementos y un rendimiento bajo al aumentar el detalle del escenario a crear.

2.2 Herramientas integradas en videojuegos

El nivel de realismo suele ser inferior y ofrecen miles de kilómetros de terrenos generados automáticamente. La mayor lacra de estas herramientas es que no están en el mercado, si no que se encuentran totalmente integradas dentro de un videojuego o simulador, este tipo de herramientas es nuestro objetivo, igualar su rendimiento aumentando la personalización y el realismo. Algunos ejemplos de videojuegos que tienen generadores procedurales de terreno son:



(a) Personalización que ofrece World Creator (b) Resultado final de World Creator

Figura 2.3: Terrain Composer

- **Minecraft [4], de Mojang Studios:** Es el juego más famoso del mundo y el más vendido, todo su mundo y elementos del mismo se generan proceduralmente, con un terreno cuya extensión es de 4096 millones de kilómetros cuadrados fue el juego con el mundo más grande hasta que salió el siguiente integrante de esta lista en 2016. El detalle del terreno es ínfimo (son todo bloques cuadrados) y fácilmente superable.

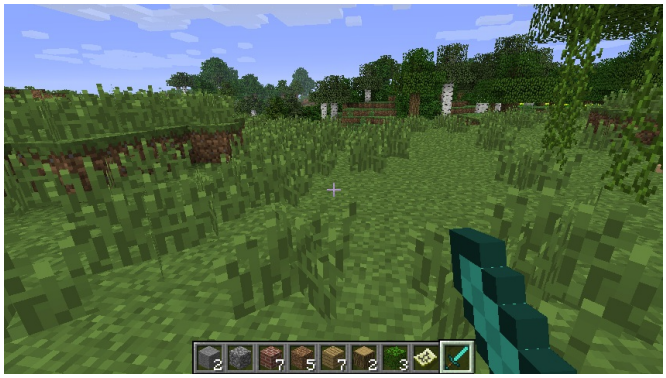


Figura 2.4: Nivel de detalle de Minecraft

- **No Man´s Sky [5], de Hello Games:** Aunque fue un fracaso como juego, sigue ostentando el récord de mapa más grande generado proceduralmente, hablamos de 18 trillones de planetas de diferentes tamaños pero generalmente entre 50 y 100 kilómetros cuadrados por planeta. Cada planeta se genera proceduralmente y el detalle del mismo es de calidad media.

- **Rust [6], de Facepunch Studios:** Es un juego que estuvo en desarrollo más de 7 años y cuya generación de terreno y entorno ha cambiado mucho a lo largo de este tiempo.

Figura 2.5: Nivel de detalle de No Man´s Sky

Actualmente genera mapa de tamaño entre 1 y 8 kilómetros cuadrados con un realismo bastante alto.

Figura 2.6: Nivel de detalle de Rust

Se detallan a continuación una serie de juegos, que integran escenarios muy grandes creados a mano y el tiempo que estuvieron en desarrollo:

- **The witcher 3:** 135 km^2 de escenarios. 3 años y medio de desarrollo con un equipo de más de 240 personas.
- **Assassin´s Creed Odyssey:** 256 km^2 de escenarios. 3 años de desarrollo con 7 equipos de unas 100 personas.
- **Just Cause 4:** 1024 km^2 de escenarios. 3 años de desarrollo con un equipo de 320 personas.
- **The Elder Scrolls 2: Daggerfall :** 161.594 Km^2 de escenarios. Casi 10 años de desarrollo.

2.3 Conclusión del estudio

En la tabla 2.1 se pueden ver los diferentes aspectos tenidos en cuenta en este estudio.

El tamaño de generación define cuantos km^2 de terrenos y decorados es capaz de generar la herramientas antes de perder rendimiento. Como se observa en la tabla, las herramientas disponibles en el mercado no soportan grandes extensiones, a menos que se invierta mucho tiempo, lo que va en contra de los objetivos del proyecto. Las herramientas integradas en videojuegos, en su mayoría no pierden eficiencia, ya que generan los elementos bajo demanda, como la herramienta desarrollada en el presente proyecto. Sin embargo, en el caso concreto del sistema de Rust, su nivel de detalle hace difícil que los elementos se generen bajo demanda, ya que su generación es muy costosa.

El tiempo de generación hace referencia al tiempo que tarda el usuario final en poder hacer uso de los elementos generados. Las herramientas disponibles tardan pocos segundos, ya que se generan en tiempo de edición, y el usuario final no percibe dicho tiempo. Las herramientas integradas en videojuegos tardan pocos segundos en la pre-configuración y después se genera en segundo plano sin que el usuario lo perciba. En el caso del sistema de Rust se genera todo de golpe, después de lo cual no hay ningún sistema de generación en segundo plano. La herramienta desarrollada en el presente proyecto tardará pocos segundos en la pre-configuración y generación inicial, tras lo cual se generarán los elementos en tiempo real.

El nivel de detalle define el realismo del entorno generado, donde las herramientas disponibles ganan por mucho, ya que los sistemas integrados en videojuegos no pueden permitirse un entorno muy detallado, ya que aumentaría el coste de generación y se perdería la generación en tiempo real. La herramienta desarrollada en el presente proyecto pretende aumentar el nivel de realismo conservando la generación en tiempo real.

El nivel de configuración hace referencia a la cantidad de parámetros que el desarrollador puede modificar para generar distintos tipos de entornos. Las herramientas disponibles tienen muchos parámetros, ya que están orientadas a la edición por parte del desarrollador. No se sabe cuántos, ni de qué tipo, son los parámetros de los sistemas integrados en videojuegos, pero el usuario final siempre percibe el mismo resultado. La herramienta desarrollada en el presente proyecto contará con un nivel de configuración alto pero no excesivo, para mantener la versatilidad y facilitar su aprendizaje.

La disponibilidad indica si el sistema puede ser usado actualmente. Las herramientas disponibles, como su propio nombre indica, pueden ser usadas por cualquier persona o empresa que pague la licencia de uso. Los sistemas integrados en videojuegos no pueden ser usados por nadie que no sea el creador de dicho videojuego. La herramienta desarrollada en el presente proyecto estará disponible en la tienda de Unity de igual manera que las herramientas disponibles.

El precio indica el nivel de inversión que se ha de efectuar para usar el sistema. Las herramientas disponibles oscilan entre 40€ y 300€, dependiendo de la calidad de los que ofrecen. Los sistemas integrados en videojuegos no están en venta. La herramienta desarrollada en el presente proyecto obtendrá un valor al finalizar el desarrollo de la misma, en función de lo que ofrezca.

Nombre	**Tamaño de generación**	**Tiempo de generación**	**Nivel de detalle**	**Nivel de configuración**	**Uso**	**Precio**
Gaia Pro	Pocos km^2	Pocos segundos (Sin colocación)	Muy alto	Muy alto	Disponible	250€
Terrain Composser	Pocos km^2	Pocos segundos (Sin colocación)	Alto	Muy alto	Disponible	45€
World Creator	Pocos km^2	Pocos segundos (Sin colocación)	Muy alto	Muy alto	Disponible	70€
Sistema de Minecraft	4.096 * 10^9 km^2	Generación en tiempo real	Muy bajo	Ninguno	No disponible	No está en venta
Sistema de No Man´s Sky	18 * 10^{20} km^2	Generación en tiempo real	Medio	Ninguno	No disponible	No está en venta
Sistema de Rust	1 - 8 km^2	3 - 5 minutos	Muy Alto	Ninguno	No disponible	No está en venta
Herramienta desarrollada en este proyecto	**10^{18} km^2**	**Generación en tiempo real**	**Alto**	**Alto**	**Disponible**	**Aún por determinar**

Cuadro 2.1: Tabla comparativa de diferentes herramientas de generación automatizada

El estudio de mercado nos arroja unos datos claros, no hay a la venta una herramienta totalmente automática, como las que integran los videojuegos vistos anteriormente, que genere grandes cantidades de terreno y los coloque adecuadamente para formar escenarios de grandes dimensiones. Que genere dichos elementos bajo demanda, con un nivel de realismo alto y sin que el usuario final perciba tiempos de carga. Por lo que el desarrollo del proyecto se considera viable.

Capítulo 3

Fundamentos

En este capítulo se analizarán las tecnologías disponibles en el mercado sobre las que se pueda desarrollar la herramienta, los conceptos básicos de la tecnología elegida y los fundamentos matemáticos sobre los que se asienta el diseño.

3.1 Tecnologias de desarrollo

3.1.1 Motores de videojuego

Un motor de videojuego es un conjunto de herramientas que facilitan el desarrollo de elementos o escenarios 3D tales como videojuegos o entornos de simulación. Su función es la de renderizar los gráficos, calcular las físicas, gestión de la memoria, inteligencia artificial, comunicación en red, etc. Suelen integrar un entorno de desarrollo que facilita las labores de codificación, compilación, ejecución y depuración, así como el diseño gráfico. Hay muchos motores de videojuego en el mercado e incluso se podría plantear la creación de uno nuevo orientado al propósito del proyecto, pero no se dispone del tiempo suficiente. De entre todos los motores de videojuego del mercado se reduce la elección a dos posibilidades, ya que ambas son gratuitas (inicialmente), tienen una amplia comunidad, poseen una tienda donde vender la herramienta al público objetivo y amplio abanico de plataformas de distribución. Estas dos opciones son Unreal Engine y Unity3D.

- **Unreal Engine [7], de Epic Games:** Fue creado en 1998 y actualmente se encuentra en su quinta versión. Usa principalmente el lenguaje de programación C + + pero actualmente está desarrollando su propio lenguaje de script optimizado para su pipeline gráfico. Está muy centrado en obtener un acabado gráfico impecable, especialmente pensado para artistas. Su aprendizaje es complejo a pesar de tener una gran comunidad. Es inicialmente gratuito pero hay que pagarle a Epic Games un 5% de todo el beneficio que se saque de su uso.

- **Unity3D [8], de Unity Technologies:** Fue creado en 2005 y actualmente se encuentra en su novena versión llamada unity 2020 (aunque ya existe una décima versión experimental llamada unity 2021). Usa principalmente el lenguaje de programación C# aunque también tiene un lenguaje de script propio y se puede usar Javascript. Es una herramienta de uso mucho más genérico, obteniendo un balance entre todos sus componentes. Su aprendizaje es muy sencillo, la propia empresa tiene cientos de tutoriales para su aprendizaje y la gran mayoría de tutoriales de internet sobre programación en entornos 3D están enfocados en esta herramienta. Es inicialmente gratuito pero si ganas más de cien mil dólares anuales derivados de sus uso debes pagar la versión Plus (400$ anuales por puesto de trabajo) y si ganas más de doscientos mil debes pagar la versión Pro (1800$ anuales por puesto de trabajo).

La decisión sobre qué motor de videojuego usar es sencilla de tomar. La principal ventaja de Unreal, los gráficos realistas, no es relevante para el desarrollo del proyecto. Las principales ventajas de Unity, el aprendizaje acelerado y mayor número de clientes potenciales, permiten, en primer lugar, dedicar más tiempo al proyecto y menos a dominar el motor de videojuego, y en segundo lugar, tener una mayor probabilidad de éxito comercial. Por lo tanto se usará Unity3D y el lenguaje de programación C#.

3.2 Fundamentos básicos de Unity

Para la realización del proyecto se utilizan ciertos componentes de Unity. Como el *Entity Component System*, necesario para aumentar la eficiencia en la generación y renderizado de los elementos decorativos. Los *GameObject*, que son los contenedores para los componentes propios de Unity (*MeshRenderer* por ejemplo) y los que se desarrollan durante el proyecto (*Mesh* que albergan el terreno que se genera, o *scripts* que contienen el comportamiento de los algoritmos).

Para una mayor comprensión de lo que el presente proyecto pretende realizar, se explicaran a continuación estos elementos de Unity.

- **ECS (*Entity Component System*[9]):** Es la tecnología orientada a datos de Unity, compuesta de tres elementos, entidades, componentes y sistemas. Las entidades son los elementos base, que pueden representar cualquier cosa en el mundo 3D, los componentes son los datos asociados a las entidades, pero organizados por los datos en sí en lugar de por entidad, y los sistemas son la lógica que transforma los datos de los componentes y consecuentemente todas las entidades que poseen dicho componente. Esto habilita que solo se actualicen los datos y no los objetos al completo, lo que reduce el coste computacional, al tiempo que habilita el *multithreading*, ya que cada componente es independiente y está aislado por lo que se puede transformar en un hilo distinto.

- ***GameObject*:** Es el objeto propio de Unity, hereda de la superclase *Object*, es la clase que representa cualquier elemento que esté dentro de una escena.

- ***Mesh*:** Los *mesh* o mallados, son el componente principal para el renderizado de cualquier elemento 3D dentro de Unity. Están formados por arrays de vértices, que son vectores de 3 dimensiones que definen los límites de un objeto. Por triángulos, que son 3 índices en el array de vértices y por diferentes arrays para la normal, color, textura y tangente de cada vértice.

- ***MeshRenderer*:** Es un componente propio de Unity, se encarga de enviar al pipeline gráfico los datos de un *Mesh* y un material, para así poder renderizarlo en pantalla.

- ***MonoBehaviour*:** Es la clase base que ofrece Unity para definir comportamientos. Los *scripts* donde estén definidos los algoritmos extenderán esta clase y se beneficiaran de sus métodos, como *Awake*() y *Start*() que se llaman una sola vez al inicio de la ejecución, o el método *update* que se llaman en cada ciclo.

3.3 Fundamentos matemáticos

Para la generación del entorno necesitaremos hacer uso de varios principios matemáticos y aplicarlos en nuestro código.

El principal será el uso del ruido, entendiéndose este como la generación de una onda n-dimensional compuesta por k ondas (funciones periódicas como el seno o coseno). Un ejemplo de función compuesta por 4 ondas sería:$f(x) = sin(x) + sin(x * 2) + sin(x * 4)/1.5 + sin(x * 20)/10$. Donde cada onda es una función seno con distinta frecuencia y amplitud. Para simplificar los cálculos y perder la periodicidad, las ondas se evalúan a intervalos, para después interpolar linealmente dichos valores y obtener continuidad, como se ve en las figuras 3.1a, 3.1b, 3.1c y 3.1d, donde se observa, como una onda con baja frecuencia y alta amplitud ofrece pocos valores grandes y una onda con frecuencia alta y baja amplitud ofrece muchos valores pequeños.

La suma de estas ondas se puede apreciar en la figura 3.1e, donde tenemos una onda compleja que puede ser vista como el contorno de un terreno montañoso. Este tipo de ondas cumplen un principio que observamos en la naturaleza: A mayor escala, menor número de variaciones pero con mayor magnitud, a menor escala, mayor número de variaciones pero con menor magnitud. O dicho de otra forma, a altas frecuencias bajas amplitudes y viceversa. Cuanto mayor sea el número de funciones apiladas, mayor el detalle del terreno y el realismo del mismo.

Este tipo de ondas, llevadas a 2 dimensiones, ofrecerán los datos necesarios para la generación de un terreno realista y detallado.

(a) Frecuencia 2 Amplitud 128

(b) Frecuencia 4 Amplitud 64

(c) Frecuencia 16 Amplitud 16

(d) Frecuencia 64 Amplitud 4

(e) Suma de las ondas anteriores

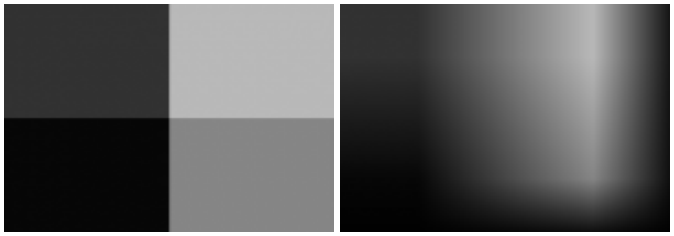
Figura 3.1: Composición de ondas con diferentes amplitudes y frecuencias

3.3.1 Ruido Perlin

El ruido perlin consiste en una función matemática que utiliza interpolación entre un gran número de gradientes pre-calculados de vectores, que construyen un valor que varía pseudo-aleatoriamente. Resultó del trabajo de Ken Perlin [10], que lo inventó para generar las texturas de la película Tron.

El resultado de esta función matemática será la altura del terreno. Dado que el terreno es tridimensional, será necesario un mapa de ruido en 2 dimensiones, que dados 2 valores devuelva un tercero que se usará como altura.

Para obtener un mapa de ruido en 2 dimensiones, se añade otra variable a la función de ruido y se evalúa en ambas direcciones (ver figura 3.2a, donde se otorga un color a cada cuadrante en función del valor devuelto). A continuación se hace una interpolación bilineal, es decir, se interpola linealmente en ambas direcciones, como se ve en la figura 3.2b. Esto ofrece un gradiente de valores más detallado con un menor número de evaluaciones, como la representada en la figura 3.2a, con una frecuencia de tan solo 4 (evaluada 4 veces con dos valores distintos para cada variable o dirección).



(a) Matriz de una onda con frecuencia 4

(b) Interpolación bilineal de la matriz anterior

Figura 3.2: Generación de un mapa de ruido Perlin en 2D

Para la realización del proyecto se va a usar la función *PerlinNoise* de la librería *Mathf* de Unity [11], pues ya tiene implementado el ruido Perlin y optimizado para la ejecución dentro del motor de videojuego. Dicha función genera un mapa en 2D como el descrito anteriormente y como se puede ver en la figura 3.3. Este mapa, aunque pseudoaleatorio, todavía sigue siendo repetitivo, por lo que se evaluará la función cíclicamente de acuerdo al nivel de detalle requerido, con distintos valores de frecuencia y amplitud en función del tipo de terreno que se pretenda generar. Dichas evaluaciones se apilarán para conformar el mapa final, cuyo grado de aleatoriedad y detalle será superior.

Figura 3.3: Ejemplo de mapa 2D generado por la función PerlinNoise de Unity

El ruido también se usará para definir los biomas y ecosistemas. Con las mismas coordenadas y otra función de ruido, que varíe en frecuencias y amplitudes, se obtiene un conjunto de valores distinto que, al aplicarles un filtro de rango (ver figura 3.4), devuelven una clasificación que podría definir tipo de vegetación, clima, fauna, etc. Dentro de cada ecosistema se podría aplicar sucesivamente nuevas funciones de ruido para dar más variabilidad.

3.3.2 Muestreos de Poisson

Una vez creado el terreno se deberá generar los decorados de cada bioma, necesarios para el realismo del entorno. Como estos decorados son objetos sólidos, hay que asegurar que no se generen en las mismas coordenadas, o lo suficientemente cercanas para que se intersequen. Para esto se usarán los muestreos de Poisson, detallados a continuación:

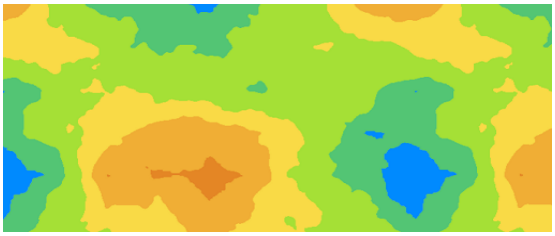


Figura 3.4: Valores de un mapa de ruido filtrados en 6 rangos

En el muestreo en disco de Poisson, el conjunto de elementos (en nuestro caso los decorados) se distribuyen en el espacio de manera densa y uniforme, pero aleatoria e irregular, lo que aumenta el realismo. La distribución comienza siendo totalmente aleatoria, pasando después un filtro de distancias, donde se comprueba que cada elemento no está en el radio de otro, si no lo está pasa a formar parte de la muestra. Dicho filtro genera una carga computacional muy elevada, ya que con cada muestra comprueba la distancia con todas las demás, por lo que un conjunto de n muestras supondrá n*(n-1) cálculos. Esta complejidad computacional es demasiado elevada para el propósito de este proyecto.

El muestreo en disco es una variante del muestreo de Poisson, donde cada elemento se somete a un ensayo de Bernoulli independiente, que determina si pasa a formar parte de la muestra. El ensayo de Bernoulli es un experimento aleatorio con dos resultados posibles, éxito o fracaso, en el que la probabilidad de éxito es la misma cada vez que se realiza. En el caso del disco de Poisson, el experimento es el filtro de distancia, que concluye con un éxito, el elemento está fuera del radio de cualquier otro elemento y pasa a formar parte de la muestra, o el fracaso, el elemento está dentro del radio de algún otro y no formará parte de la muestra.

Hay diferentes algoritmos como, por ejemplo, el creado por Cem Yuksel [12], que aumentan la eficiencia del disco de Poisson. En el presente proyecto se abordará la elaboración de una solución alternativa y eficiente para la colocación de elementos en el terreno, basada en el muestreo en disco de Poisson.

Para elaborar el conjunto de elementos iniciales que se someterán al experimento, se usará una función de ruido con una salida acotada al intervalo $[0, 1]$, se establecerá un umbral U a partir del cual se añade un elemento al conjunto (por ejemplo, $U > 0.95$ para una probabilidad muy baja, $U > 0.1$ para un probabilidad muy alta) y se establecerá una frecuencia para determinar el solapamiento entre elementos (si la función devuelve máximos cada metro se solaparán mucho, si devuelve máximos cada kilómetro se solaparán poco). Este conjunto de elementos será siempre el mismo para el mismo valor de semilla, lo que garantiza la reproducibilidad de los datos.

Capítulo 4

Metodología y planificación

4.1 Metodología utilizada

El proyecto será desarrollado siguiendo un ciclo de vida de prototipado en espiral [13, 14] en el que, en cada una de sus iteraciones, habrá que volver a las tareas *core* del mismo para refinar su diseño. Cada una de dichas iteraciones constará de análisis, diseño, implementación y pruebas. Con el uso de iteraciones se pretende obtener una herramienta entregable al final de cada ciclo y añadirle más funcionalidades y rendimiento en el siguiente incremento. La metodología está adaptada al proyecto, en cada ciclo se usarán las fases detalladas a continuación:

- **Analisis:** Se detallan los requisitos que debe cumplir la herramienta y los algoritmos al final de la iteración, así como los diagramas de flujo y de casos de uso.

- **Diseño:** Se detallan los diagramas de clases y los componentes necesarios para el funcionamiento deseado.

- **Implementación:** En esta fase se desarrollaran los algoritmos de acuerdo a la fase de diseño.

- **Pruebas:** En esta fase se harán pruebas para buscar errores de implementación o falta de funcionalidad en los componentes.

Esta metodología ha sido escogida porque permite una planificación sencilla. Así mismo se comprueba periódicamente el funcionamiento de la herramienta y se lleva un registro de los requisitos cumplidos. Por último, permite reducir tiempos o costes si hay un imprevisto, minimizando los riesgos.

4.2 Iteraciones

En esta sección se detallan todas las iteraciones que se efectuaron durante el desarrollo del proyecto.

- **Primera iteración:** Durante esta iteración se implementan los algoritmos de generación de terreno y decorados en una forma básica, la mayor parte del tiempo se emplea en el aprendizaje de Unity y sus componentes.
- **Segunda iteración:** Esta iteración se enfoca en el desarrollo del algoritmo de generación del terreno, se empezó a usar la apilación de distintos mapas de ruido y se generó un *shader* para interpretar los distintos biomas y texturizarlos adecuadamente.
- **Tercera iteración:** En este ciclo se profundiza en la generación de decorados, se dedica fundamentalmente al aprendizaje de los sistemas de *Entities* y *jobs*, se crea un algoritmo basado en estos sistemas.
- **Cuarta iteración:** El desarrollo se centra en un algoritmo que permita interactuar con las entidades, ya que usan un motor de físicas distinto.
- **Quinta iteración:** Se centra en optimizar los algoritmos y potenciar el *multithreading* para elevar el rendimiento lo máximo posible.

4.3 Planificación inicial

Para la planificación del proyecto se usó la herramienta gratuita online Canva [15], que nos permite hacer cualquier tipo de diagrama, también lo usaremos más adelante para hacer los diagramas de clase y flujo. Como se puede ver en el diagrama de Gantt 4.1, la planificación inicial estimada es de 84 días, ajustada con un margen de 7 días de la fecha de entrega del proyecto, el 23 de Junio, el inicio del proyecto fue el 24 de marzo, unos días después de la entrega del anteproyecto.

Al final de la planificación se hizo un segundo diagrama de Gantt con el desglose detallado de las tareas (ver figura 4.2).

4.3.1 Planificación económica

Junto a la planificación temporal se hace una planificación económica, basada en la estimación de tiempo y los costes por hora de los diferentes perfiles profesionales requeridos para el correcto desarrollo del proyecto, amortización de los equipos informáticos usados, licencias y formación.

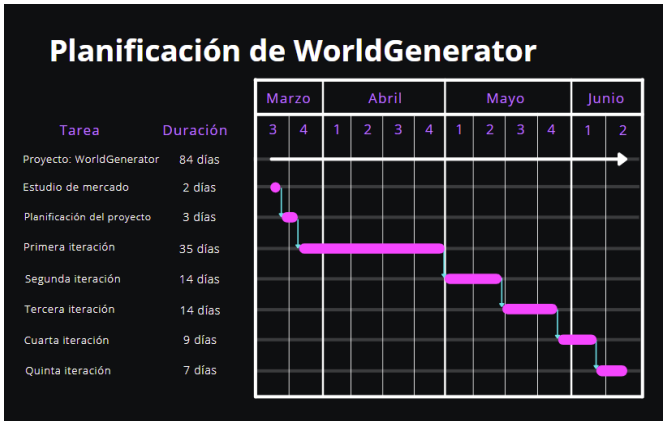


Figura 4.1: Diagrama de Gantt inicial

Perfil	Salario bruto medio anual (€)	Coste(€/h)
Analista	28.420	15,78
Programador Junior	19.217	10,67
Tester	25.734	14,29

Cuadro 4.1: Costes de cada perfil

Para el cálculo del coste de los diferentes perfiles profesionales (ver tabla 4.1), se han tomado los datos de los salarios medios en España de la página Indeed [16, 17, 18]. En la tabla 4.2 podemos ver el número de horas dedicadas por cada perfil junto al coste total de salarios.

Debido a que durante el desarrollo no se comercializa la herramienta podemos usar la licencia gratuita de Unity, junto con el uso de software libre, no supone ningún coste de licencias. La amortización del ordenador se ha calculado según la vida útil y el coste del mismo, suponiendo una jornada laboral de 8 horas diarias durante 260 días al año y una vida útil en gama alta de 6 años, tenemos un coste de 0,16€/ hora. Todos los costes citados en este apartado pueden visualizarse en la tabla 4.3.

Finalmente en la tabla 4.4 se ofrece la estimación inicial de costes del proyecto que asciende a 8.144,88 €.

Perfil	Tiempo dedicado (h)	Coste(€/h)	Coste final(€)
Analista	160	15,78	2.508,80
Programador Junior	336	10,67	3.585,12
Tester	136	14,29	1.943,44
Coste total salarios			**8.037,36**

Cuadro 4.2: Costes salariales

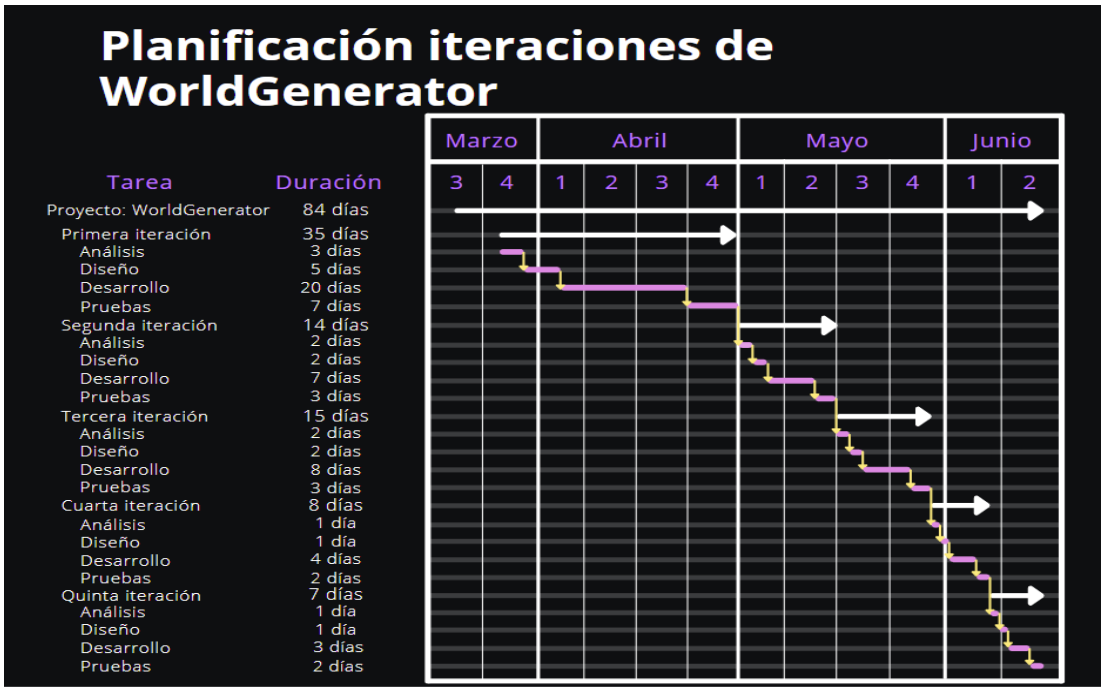


Figura 4.2: Diagrama de Gantt con iteraciones detalladas

Material/intangible	Tiempo dedicado (h)	Coste(€/h)	Coste final(€)
Equipo altas prestaciones	672	0,16	107,52
Unity 3D	472	0	0
Visual Studio 2017 Community	472	0	0
Canva	200	0	0
Coste total materiales e intangibles			**107,52**

Cuadro 4.3: Costes materiales e intangibles

Total salarios	8.037,36 €
Total materiales e intangibles	107,52 €
Total proyecto	**8.144,88 €**

Cuadro 4.4: Coste total proyecto

Capítulo 5

Desarrollo

5.1 Especificación de requisitos

La herramienta tendrá que cumplir con los siguientes requisitos funcionales:

- **Generación del terreno:** Generar un terreno a partir de unos parámetros dados, que sea realista, continuo y replicable.
- **Generación de biomas:** Poder definir el número de biomas, su frecuencia, altura media, densidad de elementos decorativos, ecosistemas de los que está compuesto y detalle del mismo.
- **Colocación de elementos:** Generar elementos decorativos propios de cada ecosistema, colocarlos naturalmente en el terreno y de forma uniforme al tiempo que irregular. Dichos elementos no se pueden solapar.
- **Creación y destrucción de elementos:** Crear y destruir, tanto el terreno como los decorados, en función de la distancia con la cámara que renderiza el entorno en pantalla.
- **Colocación de estructuras:** Poder añadir edificios o estructuras, que se coloquen en lugares apropiados y que el terreno se adapte a ellos.

Al mismo tiempo se tendrán en cuenta los siguientes requisitos no funcionales:

- **Rendimiento del renderizado:** La herramienta no debe ralentizar en demasía la escena en la que se implemente.
- **Realismo del entorno:** El terreno ha de ser lo suficientemente detallado y los elementos decorativos deben estar bien colocados.
- **Tiempo de generación:** El entorno se debe generar rápidamente, para que el sistema final tenga un tiempo de respuesta bajo.

5.2 Desarrollo Iterativo

A continuación se detalla el trabajo realizado en cada una de las iteraciones realizadas.

5.2.1 Primera Iteración

La mayor parte del tiempo de desarrollo ha sido empleado en esta iteración, ya que contiene el grueso del análisis, diseño y definición de los algoritmos, así como las pruebas pertinentes para asegurar un buen funcionamiento. Todo ello para establecer la base sobre la que poder desarrollar las siguientes iteraciones de manera más ágil y efectiva.

5.2.1.1 Análisis

Para la realización de esta iteración se planteó la creación de un sistema extensible, capaz de incorporar todas las funcionalidades necesarias en las siguientes iteraciones. También se buscó la generación realista del terreno, ya que debía ser la base para la colocación de los decorados.

Para tal propósito se definen dos módulos que albergarán los datos necesarios, además de servir como interfaz para el usuario, ya que será donde se introduzcan los datos de personalización. El primero de ellos tendrá la lógica del entorno, las dimensiones del terreno, rango de altura, punto origen o semilla y tamaño de fragmento de terreno. El segundo tendrá la lógica de creación del terreno, su contorno, biomas, ecosistemas, texturas, distribución de decorados y referencias a los modelos 3D de los decorados.

Para la creación del entorno, el módulo con la lógica del entorno (en adelante módulo Entorno), creará tantos módulos de lógica del terreno (en adelante módulo Terreno) como haya especificado el usuario en la interfaz, asignándolos a una matriz cuadrada, otorgándoles los datos necesarios para que se posicionen correctamente, tengan un rango de semilla continuo y un rango de altura. Dichos módulos Terreno usarán los datos recibidos para, generar el *mesh* con la forma del conjunto de mapas de ruido, correspondientes a los diferentes biomas que componen el terreno, asignar colores a los vértices y crear las distribuciones de elementos decorativos en función de los citados biomas.

A grandes rasgos el módulo Entorno genera n módulos Terreno, siendo $n = (\text{Tamaño del Entorno } / \text{Tamaño de Terreno})^2$. Así para un tamaño de terreno de 250 m^2 y un tamaño de entorno de 1000 m^2, tendrá que generar 16 módulos Terreno. El funcionamiento e interacción entre estos dos módulos se puede observar en el diagrama de flujo de la figura A.1. En dicha figura vemos que cada trozo de terreno es independiente de los demás, siendo el módulo Entorno quien controla los datos que le otorga a cada uno para que mantengan la continuidad. A continuación se detallan las principales fases del proceso:

El módulo Entorno obtiene los datos de generación a través del usuario, define un bucle para obtener los módulos Terreno necesarios para cubrir el tamaño del entorno.

1. **Generar fragmento de terreno:** En cada iteración se crea un módulo Terreno pasándole los datos necesarios para garantizar la continuidad. Rango de altura, coordenadas para posicionamiento y semilla para las funciones de ruido.

2. **Generar vértices de acuerdo a mapas de ruido:** El módulo Terreno recién creado toma los datos recibidos y comienza la generación del terreno. A partir de las coordenadas se posiciona en un punto del espacio tridimensional, a partir del tamaño crea las variables necesarias para albergar los datos calculados, por cada metro se crea un vértice y se evalúa la función de ruido correspondiente para otorgarle altura.

3. **Generar biomas y distribución de ecosistemas:** En función de la altura generada en el paso anterior se subdivide el terreno en biomas, siendo el más alto las montañas y el más bajo el mar. Dentro de cada bioma se evalúa una nueva función de ruido con diferentes parámetros pero misma semilla, que garantice la continuidad y reproducibilidad de los datos, que subdivide el bioma en ecosistemas.

4. **Generar distribución de decorados en función del ecosistema:** Cada uno de los ecosistemas definidos en el apartado anterior tiene un tipo de vegetación y orogenia distinta, por lo tanto hay que distribuir los decorados en función de estos. Se toma una nueva función de ruido con nuevos parámetros pero misma semilla, que garantice la continuidad y reproducibilidad de los datos, donde los máximos de dicha función representan puntos de generación, que serán sometidos a una muestreo en disco de Poisson que determine si son puntos válidos que pertenecen a la muestra. Por cada ecosistema se genera una distribución.

Cuando el módulo Terreno finaliza sus tareas se queda en modo pasivo, como un almacén de datos que el motor gráfico usa para renderizar el terreno. El módulo Entorno continúa su ejecución hasta que crea todos los módulos Terreno.

Los parámetros del módulo Entorno necesarios para la ejecución (todos Integer en el rango (-2.147.483.648, 2.147.483.647) salvo el último) son los siguientes:

- **SeedX:** Representa el punto origen del mapa de ruido en dirección horizontal.
- **SeedZ:** Representa el punto origen del mapa de ruido en dirección vertical.
- **SizeX:** Representa la anchura de cada fragmento de terreno.
- **SizeZ:** Representa la profundidad de cada fragmento de terreno.

- **SizeMap:** Representa el número de fragmentos de terreno que se generan en cada dirección.
- **MinHeight:** Altura mínima del terreno.
- **MaxHeight:** Altura máxima del terreno.
- **TerrainGenerator:** Módulo Terreno plantilla que se usa para crear los demás.

5.2.1.1.1 Actores

Los actores intervinientes a lo largo de las iteraciones serán los siguientes:

1. **Desarrollador:** La persona que usa la herramienta, encargada de introducir los parámetros de personalización iniciales. Debido a la naturaleza automática de la herramienta la única función de este actor es la de introducir datos.
2. **Motor de videojuego:** El sistema encargado de ejecutar los algoritmos y renderizar los resultados de los mismos en un entorno 3D.

5.2.1.1.2 Casos de uso

Los casos de uso definidos en esta iteración (ver figura A.5) serán los siguientes:

1. **Introducir datos del entorno:** El desarrollador introduce los parámetros que necesita el módulo Entorno. Este caso de uso se encuentra detallado en la tabla 5.1.
2. **Ejecutar el algoritmo:** Cuando el desarrollador le indica al motor de videojuego que empiece la ejecución, este manda una señal para que el algoritmo empiece a ejecutarse. Este caso de uso se encuentra detallado en la tabla 5.2.
3. **Renderizar resultados:** Cuando el algoritmo tiene los datos finaliza su ejecución, en el siguiente ciclo de reloj el motor de videojuego renderiza dichos resultados. Este caso de uso se encuentra detallado en la tabla 5.3.

5.2.1.2 Diseño

En este apartado se verá el diseño realizado durante la primera iteración, pensado para soportar nuevas características en las siguientes iteraciones. Como se ve en el diagrama de clases de la figura A.9, tenemos 3 clases que componen el núcleo de la herramienta y que contienen las funciones principales de generación del entorno. Este núcleo sigue el patrón Fachada [19], ya que el sistema se controla enteramente desde la clase *MapGenerator*, y el patrón Prototipo [20], ya que *MapGenerator* clona tantos *TerrainGenerator* como necesite ignorando su implementación, dicha clonación se efectúa a través de la función *Instantiate* de Unity que implementan todos los *GameObject*.

CU-1	**Introducir datos del entorno**
Descripción	El desarrollador introduce los parámetros que necesita el módulo Entorno.
Precondición	El modulo Entorno debe estar en la escena.
Poscondición	Todos los parámetros tienen valores válidos.
Actores	Desarrollador
Pasos	1. El desarrollador da valores válidos a todos los parámetros del algoritmo.

Cuadro 5.1: CU-1: Introducir parámetros de personalización

CU-2	**Ejecutar el algoritmo**
Descripción	Cuando el desarrollador le indica al motor de videojuego que empiece la ejecución, este manda una señal para que el algoritmo empiece a ejecutarse.
Precondición	El algoritmo ha de estar configurado.
Poscondición	El algoritmo ha generado todos los datos correctamente.
Actores	Motor de videojuego
Pasos	1. El motor de videojuego manda la señal de arranque a todos los sistemas y el algoritmo empieza su ejecución de forma autónoma.

Cuadro 5.2: CU-2:Ejecutar el algoritmo

CU-3	**Renderizar resultados**
Descripción	Cuando el algoritmo tiene los datos finaliza su ejecución, en el siguiente ciclo de reloj el motor de videojuego renderiza dichos resultados.
Precondición	El algoritmo ha de terminar su ejecución.
Poscondición	El terreno se muestra por pantalla.
Actores	Motor de videojuego
Pasos	1. El motor de videojuego arranca un nuevo ciclo de renderizado, obtiene los datos generados por el algoritmo y los renderiza en pantalla.

Cuadro 5.3: CU-3:Renderizar resultados

- **MapGenerator:** Clase que provee de un interfaz simple al usuario, ya que de otra manera tendría que configurar cada fragmento del terreno a mano, y desde la cual se controla todo el sistema de generación. Se encarga de hacer los cálculos de cuantos fragmentos de terreno son necesarios, en qué posición se tiene que colocar cada uno, que semilla necesitan para que el terreno sea continuo y finalmente crea dichos fragmentos y les entrega los datos que necesitan para generar el terreno de forma autónoma.
- **TerrainGenerator:** Clase que define cada fragmento de terreno y todo lo que contiene, encapsula los datos relevantes para la generación del terreno y posicionamiento de los elementos decorativos.
- **PoissonDiscSampling:** Clase auxiliar que proporciona la funcionalidad del muestreo en disco de Poisson. Se le entregan unos puntos extraídos de una función de ruido y devuelve una muestra con todos los puntos que son válidos para la generación de decorados.

5.2.1.3 Implementación

Para la implementación se necesita crear dos *GameObjects*, uno será el modulo Entorno y otro el módulo Terreno definidos durante el análisis (en adelante se denominarán *MapGenerator* y *TerrainGenerator* respectivamente), y tres *scripts* que albergarán las tres clases que componen el núcleo de la herramienta.

En la escena de Unity solo será necesario colocar el *GameObject MapGenerator*, que tendrá como componente el *script* que alberga el algoritmo *MapGenerator*, y este se encargará de instanciar tantos *GameObject TerrainGenerator* como sea preciso.

La clase *MapGenerator* contiene su lógica en el método *Awake()* que hereda de *Monobehaviour* (véase sección 3.2 para más información), este método es ejecutado en el primer ciclo de la ejecución del bucle de Unity. Todos los cálculos de los fragmentos del terreno se hacen de forma secuencial. Una vez terminado el primer ciclo todos los fragmentos de terreno están en la escena, correctamente colocados y renderizados.

La única desviación que se ha cometido, con respecto a la fase de diseño, es el añadido de una variable extra en la clase *TerrainGenerator*, un gradiente de color que se evalúa en función del ecosistema para poder visualizar los resultados de las pruebas.

5.2.1.4 Pruebas

Todas las pruebas se realizan con la misma semilla, el mismo tamaño de mapa (20 km^2) y en el mismo equipo informático (detallado en la tabla 5.4), así garantizamos consistencia en los datos medidos.

Se han realizado un total de 6 pruebas, que se detallan en la tabla 5.5.

Componente	Tipo
Procesador	AMD Ryzen 9 3900X 12-Core Processor 3.79 GHz
Memoria RAM	32 GB DDR4 3200 MHz
Tarjeta Gráfica	Nvidia GTX 1070 8GB GDDR5

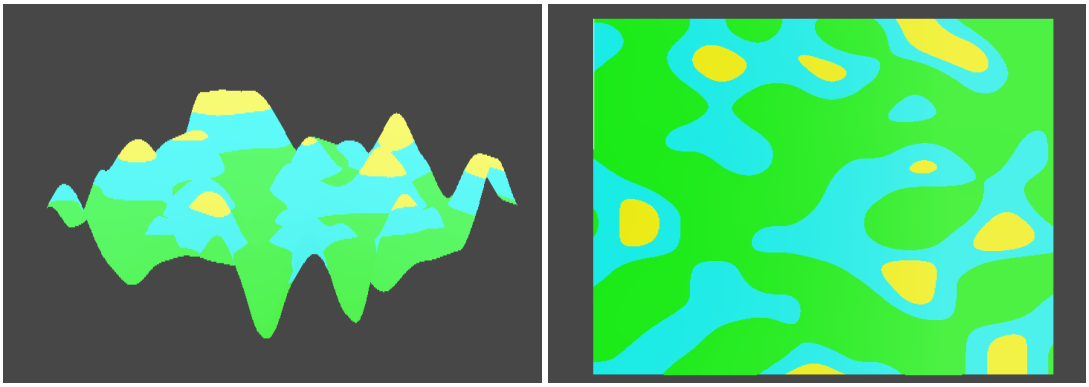
Cuadro 5.4: Características equipo informático

Cuadro 5.5: Pruebas realizadas en la primera iteración

Nº	Descripción	Resultado esperado	Resultado obtenido
1	Generación de un terreno con 3 biomas sin ecosistemas, 5 km de longitud de onda del mapa de ruido e intervalo de altura [0, 1500] metros.	Terreno compuesto de 3 biomas, con altura en rango [0, 1500] metros, sin discontinuidades y como máximo, 4 puntos de altura máxima y 4 puntos de altura mínima en cualquier fila o columna de la cuadrícula.	Terreno compuesto de 3 biomas, con altura en rango [0, 1500] metros, sin discontinuidades y como máximo, 4 puntos de altura máxima y 4 puntos de altura mínima en cualquier fila o columna de la cuadrícula. (Ver figura 5.1)
2	Generación de un terreno con 7 biomas sin ecosistemas, 10 km de longitud de onda del mapa de ruido e intervalo de altura [0, 1500] metros.	Terreno compuesto de 7 biomas, con altura en rango [0, 1500] metros, sin discontinuidades y como máximo, 2 máximos y 2 mínimos en cualquier fila o columna de la cuadrícula.	Terreno compuesto de 7 biomas, con altura en rango [0, 1500] metros, sin discontinuidades y como máximo, 2 puntos de altura máxima y 2 puntos de altura mínima en cualquier fila o columna de la cuadrícula. (Ver figura 5.2)
3	Generación de un terreno con 3 biomas y 3 ecosistemas por bioma, 5 km de longitud de onda del mapa de ruido, 1 km de longitud de onda del mapa de ruido de ecosistemas e intervalo de altura [0, 1500] metros.	Terreno compuesto de 3 biomas separados en 3 ecosistemas cada uno, con altura en rango [0, 1500] metros, sin discontinuidades, como máximo, 4 puntos de altura máxima y 4 puntos de altura mínima en cualquier fila o columna de la cuadrícula, muchísimos intercambios de ecosistema dentro de un bioma.	Terreno compuesto de 3 biomas separados en 3 ecosistemas cada uno, con altura en rango [0, 1500] metros, sin discontinuidades y como máximo, 4 puntos de altura máxima y 4 puntos de altura mínima en cualquier fila o columna de la cuadrícula, muchísimos intercambios de ecosistema dentro de un bioma. (Ver figura 5.3)

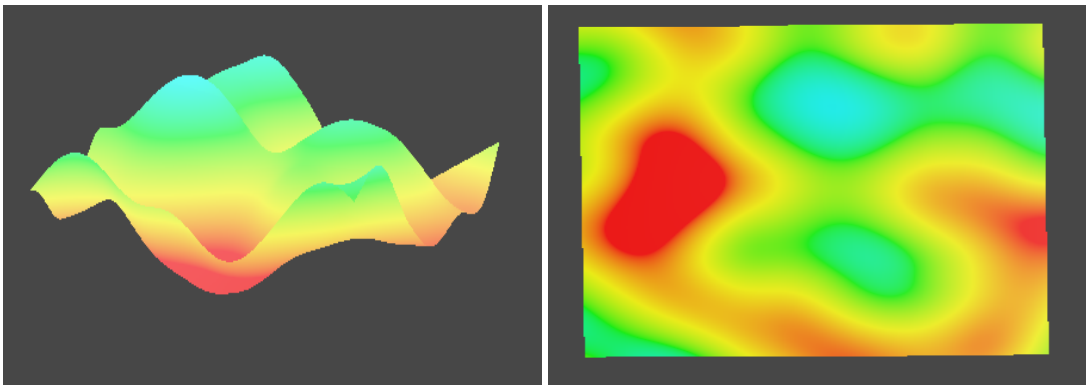
Cuadro 5.5: Pruebas realizadas en la primera iteración (continuación)

Nº	Descripción	Resultado esperado	Resultado obtenido
4	Generación de un terreno con 3 biomas y 3 ecosistemas por bioma, 15 km de longitud de onda del mapa de ruido, 2 km de longitud de onda del mapa de ruido de ecosistemas e intervalo de altura [0, 1500] metros.	Terreno compuesto de 3 biomas separados en 3 ecosistemas cada uno, con altura en rango [0, 1500] metros, sin discontinuidades, como máximo, 1 punto de altura máxima y 1 punto de altura mínima en cualquier fila o columna de la cuadrícula, muchos intercambios de ecosistema dentro de un bioma.	Terreno compuesto de 3 biomas separados en 3 ecosistemas cada uno, con altura en rango [0, 1500] metros, sin discontinuidades y como máximo, 1 punto de altura máxima y 1 punto de altura mínima en cualquier fila o columna de la cuadrícula, muchos intercambios de ecosistema dentro de un bioma. (Ver figura 5.4)
5	Generación de un terreno con 3 biomas y 3 ecosistemas por bioma, 20 km de longitud de onda del mapa de ruido, 10 km de longitud de onda del mapa de ruido de ecosistemas e intervalo de altura [0, 1500] metros.	Terreno compuesto de 3 biomas separados en 3 ecosistemas cada uno, con altura en rango [0, 1500] metros, sin discontinuidades y como máximo, 1 punto de altura máxima y 1 punto de altura mínima en cualquier fila o columna de la cuadrícula, pocos intercambios de bioma.	Terreno compuesto de 3 biomas separados en 3 ecosistemas cada uno, con altura en rango [0, 1500] metros, sin discontinuidades y como máximo, 1 punto de altura máxima y 1 punto de altura mínima en cualquier fila o columna de la cuadrícula, pocos intercambios de bioma. (Ver figura 5.5)
6	Generación de un terreno con 3 biomas y 3 ecosistemas por bioma, 20 km de longitud de onda del mapa de ruido, 10 km de longitud de onda del mapa de ruido de ecosistemas e intervalo de altura [0, 3000] metros.	Terreno compuesto de 3 biomas separados en 3 ecosistemas cada uno, con altura en rango [0, 3000] metros, sin discontinuidades y como máximo, 1 punto de altura máxima y 1 punto de altura mínima en cualquier fila o columna de la cuadrícula, pocos intercambios de bioma.	Terreno compuesto de 3 biomas separados en 3 ecosistemas cada uno, con altura en rango [0, 1500] metros, sin discontinuidades y como máximo, 1 punto de altura máxima y 1 punto de altura mínima en cualquier fila o columna de la cuadrícula, pocos intercambios de bioma. (ver figura 5.6)



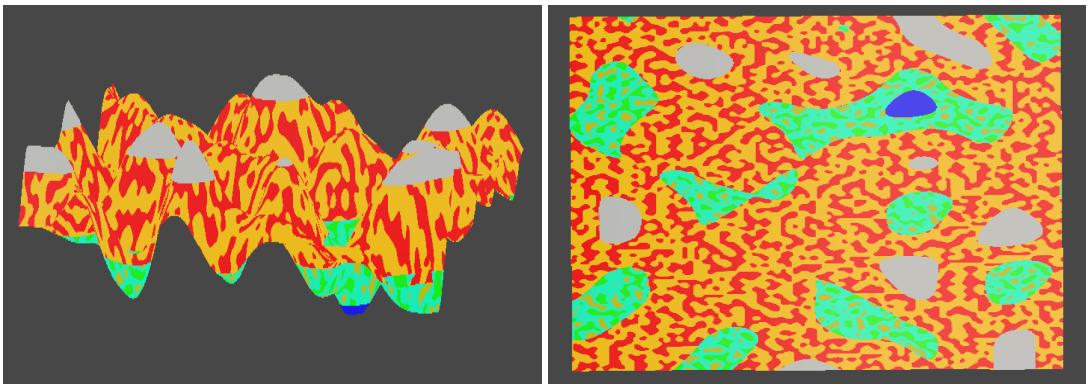
(a) Vista lateral (b) Vista de pájaro

Figura 5.1: Resultados de la primera prueba



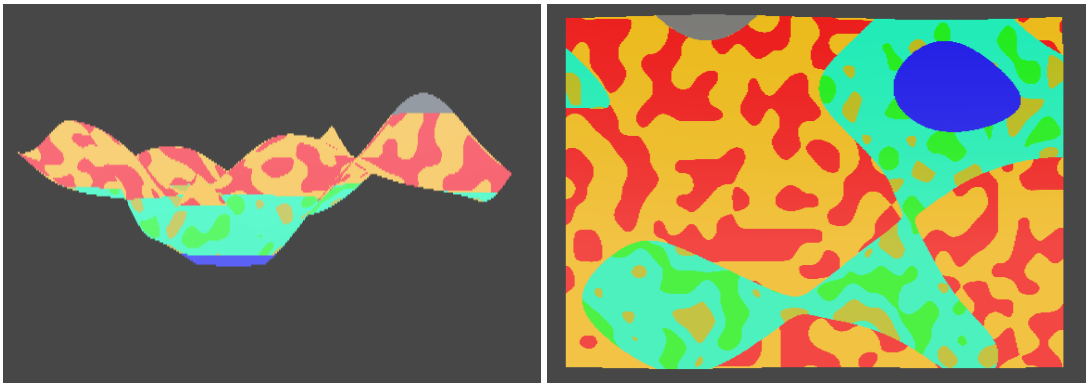
(a) Vista lateral (b) Vista de pájaro

Figura 5.2: Resultados de la segunda prueba



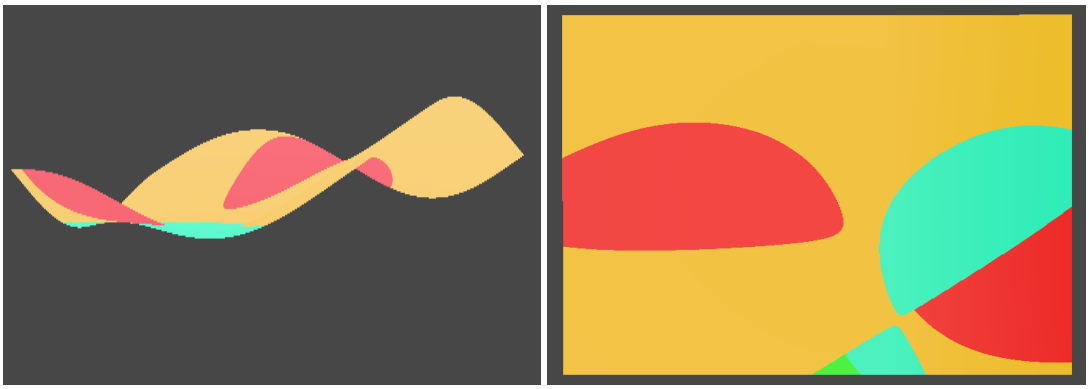
(a) Vista lateral (b) Vista de pájaro

Figura 5.3: Resultados de la tercera prueba



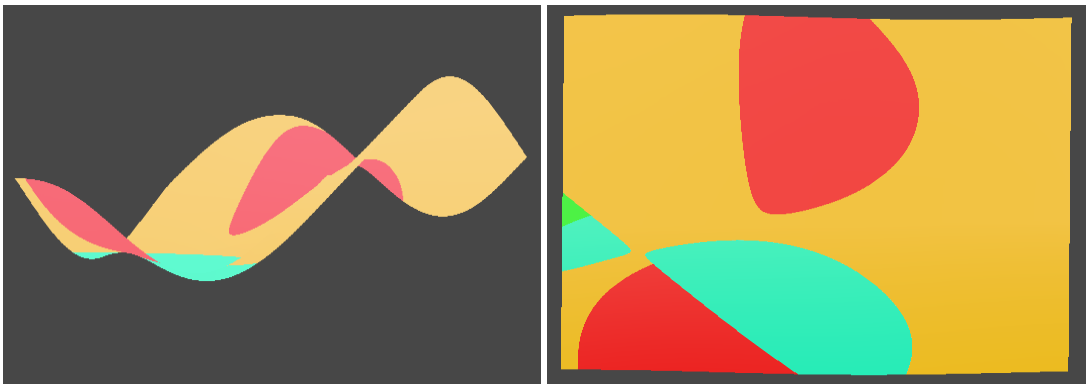
(a) Vista lateral (b) Vista de pájaro

Figura 5.4: Resultados de la cuarta prueba



(a) Vista lateral (b) Vista de pájaro

Figura 5.5: Resultados de la quinta prueba



(a) Vista lateral (b) Vista de pájaro

Figura 5.6: Resultados de la sexta prueba

El resultado de las pruebas no revela ningún error que deba ser corregido. Todas las ejecuciones tardan entre 25 y 35 segundos, después de ese tiempo no se requiere ningún cálculo a mayores. El impacto en el rendimiento total del entorno 3D es bajo ya que todavía no hay detalles ni generación de decorados.

El resultado de la iteración en general es satisfactoria, el núcleo de la herramienta funciona y está listo para incorporar las características de las siguientes iteraciones. Contiene todos los datos necesarios para escalar y ofrecer un detalle alto y realismo.

5.2.2 Segunda Iteración

En esta iteración el desarrollo se centra en el realismo del terreno y la eficiencia en el tiempo de generación. Se incluirá el sistema de creación y destrucción de elementos, basado en la distancia de cada fragmento de terreno a la cámara que renderiza el mundo 3D en pantalla. También se implementa la personalización del terreno, el aumento de realismo del mismo y un *shader* que interprete los biomas y ecosistemas generados para aplicarles una textura.

5.2.2.1 Análisis

Para la realización de esta iteración se modifica el módulo Terreno, creado en la iteración anterior, para adecuar su funcionamiento a un modo asíncrono, que no bloquee la ejecución al tiempo que la habilita bajo demanda. Dicha demanda será controlada por un nuevo módulo (en adelante módulo Activador), que se comunicará con cada módulo Terreno cuando la distancia entre ambos baje de un umbral, indicándoles que comiencen a hacer los cálculos del fragmento de terreno que deben representar. Si la distancia baja de un segundo umbral les indicará que ponga sus datos visibles para que el motor gráfico los renderize en pantalla.

A grandes rasgos, el modulo Activador establece un perímetro de cálculo y un perímetro más pequeño de visualización. Los módulos Terreno que entren en el perímetro de cálculo generan su trozo de terreno en segundo plano y lo mantienen oculto. Cuando un módulo Terreno cruza el perímetro de visualización, pone sus datos visibles y se renderizan en pantalla si la cámara apunta en su dirección.

El módulo Entorno sigue con el mismo comportamiento, que será una constante durante todo el desarrollo. Realiza las mismas acciones, pero los módulos Terreno ahora solo almacenan las variables cuando son creados.

El funcionamiento e interacción entre los módulos se observa en el diagrama de flujo de la figura A.2. A continuación se detallan las nuevas fases del proceso:

1. **Entrada en el perímetro de cálculo:** Evento disparador que se activa si un módulo Terreno entra en el perímetro de cálculo, tras lo cual el modulo Activador le indica al mismo que inicie sus cálculos en segundo plano.

2. **Salida del perímetro de cálculo:** Evento disparador que se activa si un módulo Terreno sale del perímetro de cálculo, tras lo cual el modulo Activador le indica al mismo que elimine sus datos.

3. **Entrada en el perímetro de visualización:** Evento disparador que se activa si un módulo Terreno entra en el perímetro de visualización, tras lo cual el modulo Activador le indica al mismo que ponga sus datos visibles.

4. **Salida del perímetro de visualización:** Evento disparador que se activa si un módulo Terreno sale del perímetro de visualización, tras lo cual el modulo Activador le indica al mismo que oculte sus datos.

5. **Generar vértices de acuerdo a mapas de ruido, Generar biomas y distribución de ecosistemas y Generar distribución de decorados en función del ecosistema:** Estas 3 fases conservan su comportamiento, pero su ejecución pasa a ser asíncrona para evitar el bloqueo del sistema.

6. **Destrucción de datos y liberación de memoria:** El módulo Terreno correspondiente elimina sus datos. El recolector de basura se encargará de liberar la memoria de los datos eliminados.

7. **Publicar datos para el renderizado:** El módulo Terreno correspondiente pone sus datos visibles. En el siguiente ciclo de renderizado, el motor gráfico tomará esos datos para mostrar el fragmento de terreno.

8. **Ocultar datos para el renderizado:** El módulo Terreno correspondiente oculta sus datos. El motor gráfico no renderiza el fragmento de terreno lo que ahorra recursos y mejora el rendimiento.

5.2.2.1.1 Actores

Los actores intervinientes a lo largo de las iteraciones serán los mismos que en la primera iteración (véase sección 5.2.1.1.1).

5.2.2.1.2 Casos de uso

Los casos de uso añadidos en esta iteración (ver figura A.6) serán los siguientes:

1. **Introducir parámetros de personalización:** El desarrollador introduce los parámetros de personalización del módulo Terreno. Este caso de uso se encuentra detallado en la tabla 5.6.

2. **Activar disparadores:** El motor de videojuego activa los disparadores en función de la distancia entre cada módulo Terreno con el modulo Activador. Este caso de uso se encuentra detallado en la tabla 5.7.

CU-4	**Introducir parámetros de personalización**
Descripción	El desarrollador introduce los parámetros de personalización del módulo Terreno plantilla.
Precondición	El módulo Terreno plantilla debe estar asignado al módulo Entorno.
Poscondición	Todos los parámetros tienen valores válidos.
Actores	Desarrollador
Pasos	1. El desarrollador da valores válidos a todos los parámetros del módulo Terreno plantilla.

Cuadro 5.6: CU-4: Introducir parámetros de personalización

CU-5	**Activar disparadores**
Descripción	El motor de videojuego activa los disparadores en función de la distancia entre cada módulo Terreno con el modulo Activador.
Precondición	Los módulos Terreno deben estar creados con sus variables de generación asignadas.
Poscondición	El módulo Terreno ha efectuado la acción que el modulo Activador le ha solicitado.
Actores	Motor de videojuego
Pasos	1. El motor de videojuego activa un disparador si un módulo Terreno entra o sale de alguno de los perímetros del módulo Activador. 2. El modulo Activador captura el disparador y, en función de este, solicita una acción al módulo Terreno.

Cuadro 5.7: CU-5: Activar disparadores

5.2.2.2 Diseño

El diseño realizado durante la segunda iteración, que añade características y funcionalidad, puede verse en la figura A.10. Se añade la clase *Biome* que contiene todas las características de biomas y ecosistemas, dichas características son necesarias para una correcta personalización del terreno y se detallan a continuación:

1. **Identifier:** Identificador del bioma, necesario para diferenciar un bioma de otro y para acceder a las características de cada uno en concreto.

2. **biomeDetail:** Número de mapas de ruido que se apilan para obtener el resultado final.

3. **biomeFrequency:** Frecuencia inicial de repetición del bioma, número de veces que el valor del mapa de ruido oscila entre el máximo y el mínimo.

4. **frequencyMultiplier:** Multiplicador que se aplica, en cada ciclo de detalle, a la frecuencia del bioma, para disminuir la distancia entre máximos y mínimos en el mapa de ruido de la iteración.

5. **maxBiomeHeight:** Porcentaje de la altura máxima del entorno que representa la altura máxima del bioma. Es la amplitud inicial del bioma, que define el máximo y el mínimo del mapa de ruido.

6. **amplitudeDivisor:** Divisor que se aplica, en cada ciclo de detalle, a la amplitud del bioma, para disminuir la distancia entre los máximos y mínimos del mapa de ruido de la iteración.

7. **biomeCurve:** Curva paramétrica que representa el contorno de las formas geológicas que aparecerán en el terreno. Es un multiplicador extra que se aplica al mapa de ruido para forzar formas concretas de terrenos.

8. **ecosystemFrequency:** Frecuencia de repetición de los ecosistemas que componen el bioma. A mayor frecuencia ecosistemas más pequeños y a menor frecuencia ecosistemas más grandes.

9. **ecosystemsNames:** Lista con los nombres de los ecosistemas que componen el bioma. Se usa para conocer el número de ecosistemas y se usará para acceder a los decorados de cada ecosistema.

La clase *TerrainGenerator* incorpora dos nuevos parámetros:

- **biomesFrecuency:** Controla cómo se reparten los biomas en el entorno y su extensión.

- **biomes:** Lista con los biomas que componen el terreno. Cada bioma es una clase *Biome* con sus propias características.

La clase *TerrainGenerator* incorpora cuatro nuevos métodos públicos que junto a las clases *ActivateCalculus* y *ActivateVisualization*, que se corresponden con el modulo Activador, y las clases propias de Unity conforman un patrón Observador [21] que se puede ver en la figura A.11.

El *Collider* que implementan *ActivateCalculus* y *ActivateVisualization* hace la función de observador, cuando el *Collider* que implementa un *TerrainGenerator* entra o sale de su perímetro, hace una llamada a las funciones *OnTriggerEnter* o *OnTriggerExit* del observador concreto oportuno (*ActivateCalculus* o *ActivateVisualization*), que tienen estos métodos sobrescritos para llamar a las funciones de *TerrainGenerator* explicadas a continuación:

1. **InRangeCalculus:** *TerrainGenerator* comienza a realizar los cálculos correspondientes a su fragmento de terreno.

2. **OutRangeCalculus:** *TerrainGenerator* elimina los datos referidos a su fragmento de terreno para liberar memoria.

3. **InRangeVisualization:** *TerrainGenerator* pública los datos de su fragmento de terreno para que puedan ser renderizados.

4. **OutRangeVisualization:** *TerrainGenerator* pone los datos de su fragmento de terreno ocultos para evitar que se renderizen.

Se define un *shader* usando la herramienta de nodos *ShaderGraph* de Unity, este contiene la lógica para evaluar el color asignado a cada vértice y texturizarlo correspondientemente. Contiene dos parámetros principales, un gradiente, que ha de ser el mismo usado en la clase *TerrainGenerator*, y un número de ecosistemas, que ha de ser el mismo número de ecosistemas que los definidos en la clase *TerrainGenerator*. El funcionamiento del *shader* se puede observar en la figura 5.7, y se explican sus pasos a continuación:

- Primero se divide 0.99 entre el número de ecosistemas para obtener el ciclo de evaluación del gradiente.

- Se suma el ciclo al resultado de la suma de ciclo anterior. En el primer caso se suma a 0.

- Se evalúa el gradiente con el valor obtenido en el paso anterior, dando como resultado un color. Este color junto al color del vértice serán las entradas de un filtro. Si son iguales la salida del filtro será 1, si son diferentes la salida será 0.

- La salida del filtro está conectada al parámetro de interpolación de un interpolador lineal, que contiene la textura del ecosistema en un extremo y la salida del interpolador anterior en el otro. Como el valor solo puede ser 1 o 0, la salida del interpolador será la textura actual o el resultado anterior. Esta configuración en cascada asegura la propagación del resultado correcto, ya que solo un filtro dará un resultado positivo.

- Cada interpolador está conectado al anterior, salvo el primero que está conectado a una textura por defecto. La salida del último está conectada a la salida que texturiza el vértice.

Como el gradiente es sólido ningún valor se repite, por lo que si un filtro da positivo todos los demás dan negativo y solo se aplica la textura correspondiente al positivo. Si el usuario desea crear más de los 3 ecosistemas incluidos por defecto, deberá duplicar un grupo de nodos *EcosystemMask* y conectar las entradas y salidas siguiendo el mismo diseño, como se puede ver en la figura 5.7.

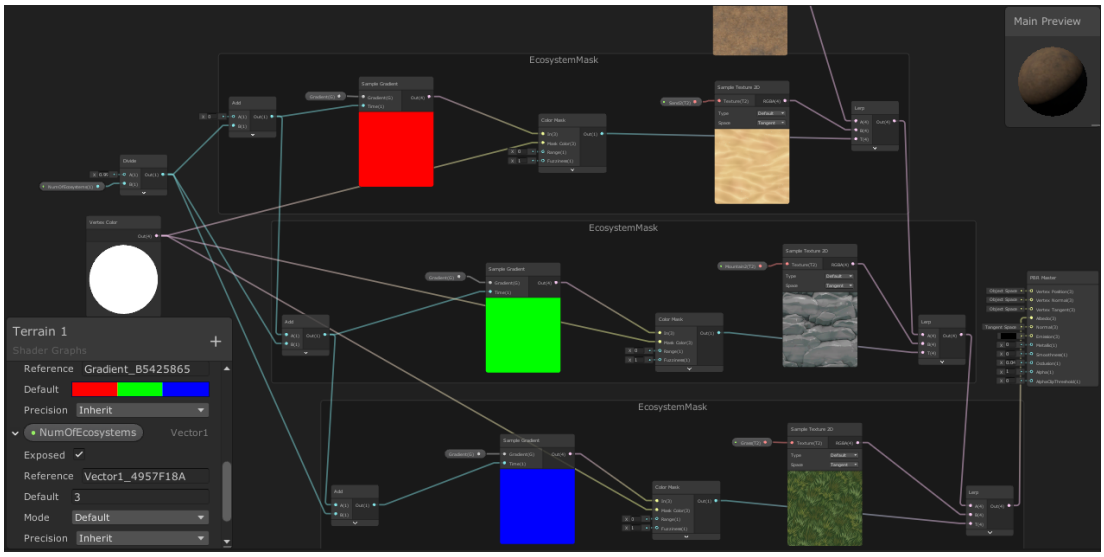


Figura 5.7: *Shader* encargado de otorgar texturas a los vértices del terreno.

5.2.2.3 Implementación

Para la implementación se necesita crear dos *GameObjects* que alberguen los *scripts* que definen las clases *ActivateCalculus* y *ActivateVisualization*. Cada *GameObject* deberá contener un componente de tipo *Rigidbody* y un componente de tipo *Collider* (se opta por el componente *SphereCollider* ya que es lo más próximo a la visión humana). Estos dos *GameObjects* serán hijos en la jerarquía de la cámara que visualiza el mundo, por lo que se moverán y rotarán en las mismas direcciones, pasando así el control del movimiento del perímetro de cálculo

y visualización a la cámara. A partir de este momento solo se generarán y visualizarán los fragmentos de terreno a una determinada distancia de la cámara.

Se creará un *script* para albergar la clase *Biome*. Este, a diferencia de todos los anteriores, no heredará de la clase *Monobehaviour* propia de Unity, ya que no necesita de sus métodos, pues solo es un contenedor de parámetros de personalización.

Se comete una desviación con respecto a la fase de diseño. El acotamiento de los parámetros de *Biome* a unos rangos que garanticen un buen funcionamiento, ya que, por ejemplo, si se pone el nivel de detalle de bioma a 1000 los cálculos serían muy complejos y se perdería la generación en tiempo real, además de que no aportarían nada, ya que la amplitud se dividiría tanto que el terreno no se elevaría ni milímetros.

5.2.2.4 Pruebas

Todas las pruebas se realizan con misma semilla, mismo tamaño de entorno (25 km^2, 5 km de lado) y el mismo equipo informático (véase tabla 5.4). Se variarán los perímetros de cálculo y visualización para comprobar el funcionamiento de la generación por distancia. Se mantendrá siempre el mismo rango de altura ([0, 2000]), ya que los biomas son los que definen sus diferentes alturas. Debido a la gran cantidad de parámetros referentes a biomas y ecosistemas, se detallaran los mismos en imágenes (véase sección 5.2.2.2 para definición de los mismos)

Se han realizado un total de 9 pruebas, detalladas en la tabla 5.8.

Cuadro 5.8: Pruebas realizadas en la segunda iteración

Nº	**Descripción**	**Resultado esperado**	**Resultado obtenido**
1	Generación de un entorno con los parámetros referidos a biomas detallados en la figura 5.8a. 1000 metros de radio de visualización y 1500 metros de radio de cálculo.	Terreno compuesto de 1 bioma y 2 ecosistemas, baja densidad de cambios de altura y nivel de detalle bajo. Solo se generan los fragmentos dentro del perímetro de visualización. Terreno continuo sin cambios bruscos de altura.	Terreno compuesto de 1 bioma y 2 ecosistemas, baja densidad de cambios de altura y nivel de detalle bajo. Solo se generan los fragmentos dentro del perímetro de visualización. Terreno continuo sin cambios bruscos de altura. (Ver figura 5.8)

Cuadro 5.8: Pruebas realizadas en la segunda iteración (continuación)

Nº	Descripción	Resultado esperado	Resultado obtenido
2	Generación de un entorno con los parámetros referidos a biomas detallados en la figura 5.9a. 1000 metros de radio de visualización y 1500 metros de radio de cálculo.	Terreno compuesto de 1 bioma y 2 ecosistemas, densidad media de cambios de altura y nivel de detalle alto. Solo se generan los fragmentos dentro del perímetro de visualización. Terreno continuo sin cambios bruscos de altura.	Terreno compuesto de 1 bioma y 2 ecosistemas, densidad media de cambios de altura y nivel de detalle alto. Solo se generan los fragmentos dentro del perímetro de visualización. Terreno continuo sin cambios bruscos de altura. (Ver figura 5.9)
3	Generación de un entorno con los parámetros referidos a biomas detallados en la figura 5.10a. 1000 metros de radio de visualización y 1500 metros de radio de cálculo.	Terreno compuesto de 1 bioma y 2 ecosistemas, baja densidad de cambios de altura y nivel de detalle alto. Solo se generan los fragmentos dentro del perímetro de visualización. Terreno continuo sin cambios bruscos de altura.	Terreno compuesto de 1 bioma y 2 ecosistemas, baja densidad de cambios de altura y nivel de detalle alto. Solo se generan los fragmentos dentro del perímetro de visualización. Terreno continuo sin cambios bruscos de altura. (Ver figura 5.10)
4	Generación de un entorno con los parámetros referidos a biomas detallados en la figura 5.11a. 1500 metros de radio de visualización y 2000 metros de radio de cálculo.	Terreno compuesto de 2 biomas y 4 ecosistemas, densidad de cambios de altura y nivel de detalle variable de acuerdo a bioma. Solo se generan los fragmentos dentro del perímetro de visualización. Terreno continuo sin cambios bruscos de altura.	Terreno compuesto de 2 biomas y 4 ecosistemas, densidad de cambios de altura y nivel de detalle variable de acuerdo a bioma. Solo se generan los fragmentos dentro del perímetro de visualización. Terreno continuo sin cambios bruscos de altura. (Ver figura 5.11)

Cuadro 5.8: Pruebas realizadas en la segunda iteración (continuación)

Nº	Descripción	Resultado esperado	Resultado obtenido
5	Generación de un entorno con los parámetros referidos a biomas detallados en la figura 5.12a. 1500 metros de radio de visualización y 2000 metros de radio de cálculo.	Terreno compuesto de 2 biomas y 4 ecosistemas, densidad de cambios de altura y nivel de detalle variable de acuerdo a bioma. Solo se generan los fragmentos dentro del perímetro de visualización. Terreno continuo sin cambios bruscos de altura.	Terreno compuesto de 2 biomas y 4 ecosistemas, densidad de cambios de altura y nivel de detalle variable de acuerdo a bioma. Solo se generan los fragmentos dentro del perímetro de visualización. Terreno continuo sin cambios bruscos de altura. (Ver figura 5.12)
6	Generación de un entorno con los parámetros referidos a biomas detallados en la figura 5.13a. 3000 metros de radio de visualización y 3500 metros de radio de cálculo.	Terreno compuesto de 2 biomas y 4 ecosistemas, densidad de cambios de altura y nivel de detalle variable de acuerdo a bioma. Solo se generan los fragmentos dentro del perímetro de visualización. Terreno continuo sin cambios bruscos de altura.	Terreno compuesto de 2 biomas y 4 ecosistemas, densidad de cambios de altura y nivel de detalle variable de acuerdo a bioma. Solo se generan los fragmentos dentro del perímetro de visualización. Terreno continuo sin cambios bruscos de altura. (Ver figura 5.13)
7	Generación de un entorno con los parámetros referidos a biomas detallados en la figura 5.14a. 1500 metros de radio de visualización y 2000 metros de radio de cálculo.	Terreno compuesto de 3 biomas y 6 ecosistemas, densidad de cambios de altura y nivel de detalle variable de acuerdo a bioma. Solo se generan los fragmentos dentro del perímetro de visualización. Terreno continuo sin cambios bruscos de altura.	Terreno compuesto de 3 biomas y 6 ecosistemas, densidad de cambios de altura y nivel de detalle variable de acuerdo a bioma. Solo se generan los fragmentos dentro del perímetro de visualización. Terreno continuo con algún cambio brusco de altura. (Ver figura 5.14)

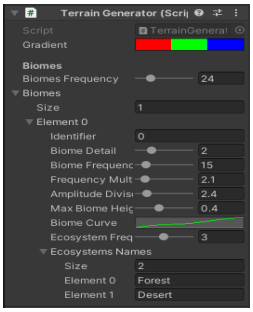
Cuadro 5.8: Pruebas realizadas en la segunda iteración (continuación)

Nº	Descripción	Resultado esperado	Resultado obtenido
8	Generación de un entorno con los parámetros referidos a biomas detallados en la figura 5.15a. 3000 metros de radio de visualización y 3500 metros de radio de cálculo.	Terreno compuesto de 3 biomas y 6 ecosistemas, densidad de cambios de altura y nivel de detalle variable de acuerdo a bioma. Solo se generan los fragmentos dentro del perímetro de visualización. Terreno continuo sin cambios bruscos de altura.	Terreno compuesto de 3 biomas y 6 ecosistemas, densidad de cambios de altura y nivel de detalle variable de acuerdo a bioma. Solo se generan los fragmentos dentro del perímetro de visualización. Terreno continuo con algún cambio brusco de altura. (Ver figura 5.15)
9	Generación de un entorno con los parámetros referidos a biomas detallados en la figura 5.16a. 1500 metros de radio de visualización y 2000 metros de radio de cálculo.	Terreno compuesto de 3 biomas y 6 ecosistemas, densidad de cambios de altura y nivel de detalle variable de acuerdo a bioma. Solo se generan los fragmentos dentro del perímetro de visualización. Terreno continuo sin cambios bruscos de altura.	Terreno compuesto de 3 biomas y 6 ecosistemas, densidad de cambios de altura y nivel de detalle variable de acuerdo a bioma. Solo se generan los fragmentos dentro del perímetro de visualización. Terreno continuo sin cambios bruscos de altura. (Ver figura 5.16)

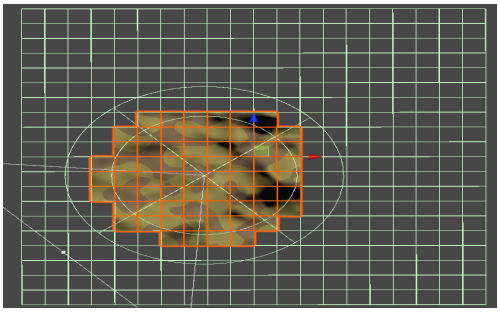
En las 3 primeras pruebas se comparten los perímetros de visualización y cálculo. También se comparte el número de biomas y ecosistemas que, junto al uso de la misma semilla, proporcionan la misma distribución de texturas. Dicha distribución se puede observar en las imágenes aéreas de las figuras 5.8, 5.9 y 5.10. En las figuras 5.9 y 5.10 se observa como la curva del bioma modifica el contorno del terreno ofreciendo un alto grado de personalización.

En la cuarta y quinta prueba se puede observar como el bioma pierde las características geológicas que lo definen según la proximidad con otros biomas, siendo estas características resaltadas a medida que se aleja el punto de unión. Este método ofrece transiciones más suaves y naturales, sin cambios bruscos de altura.

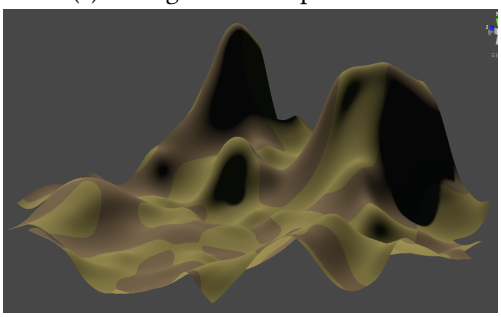
En la sexta prueba se plantea la situación en la cual los perímetros de visualización y cálculo son mayores que el tamaño del entorno. Por lo que todo el terreno es generado inicialmente con un coste computacional elevado.



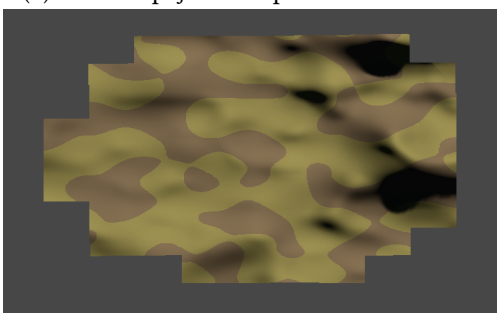
(a) Configuración de parámetros.



(b) Vista de pájaro con perímetros visibles

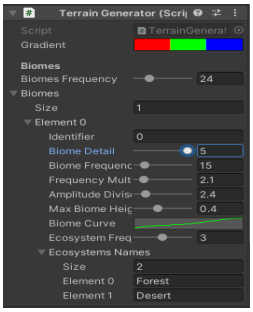


(c) Vista lateral

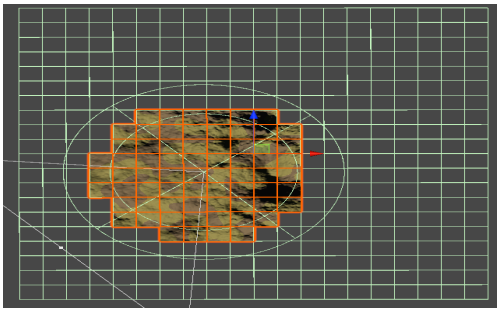


(d) Vista de pájaro.

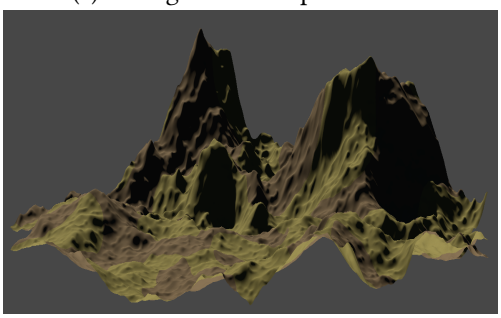
Figura 5.8: Resultados de la primera prueba



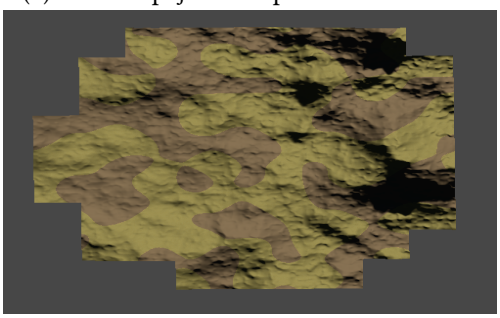
(a) Configuración de parámetros.



(b) Vista de pájaro con perímetros visibles

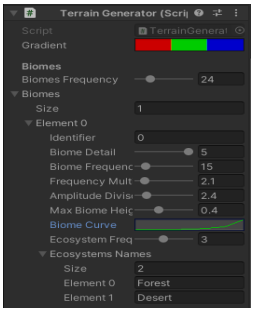


(c) Vista lateral

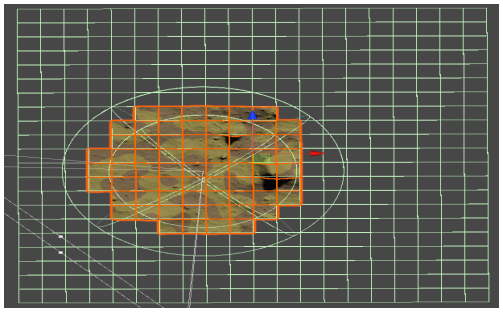


(d) Vista de pájaro.

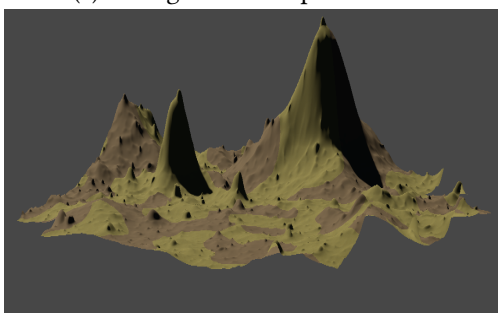
Figura 5.9: Resultados de la segunda prueba



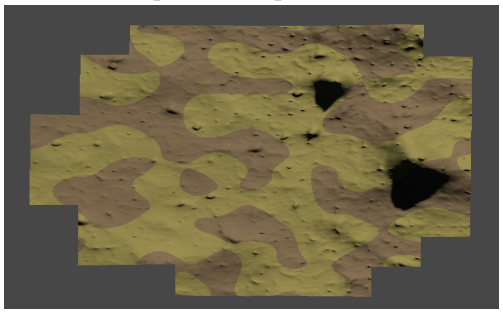
(a) Configuración de parámetros.



(b) Vista de pájaro con perímetros visibles

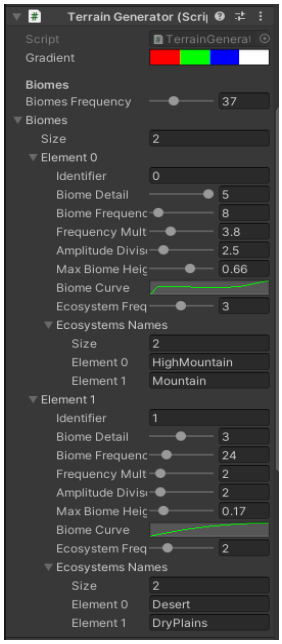


(c) Vista lateral

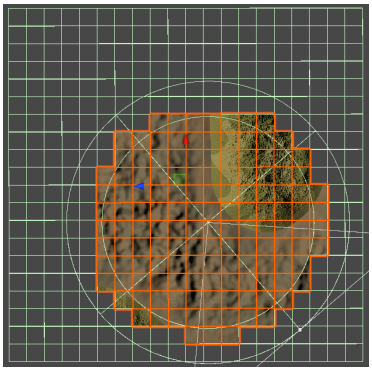


(d) Vista de pájaro.

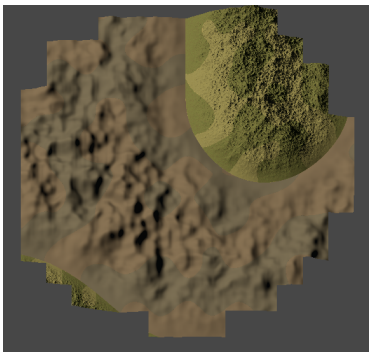
Figura 5.10: Resultados de la tercera prueba



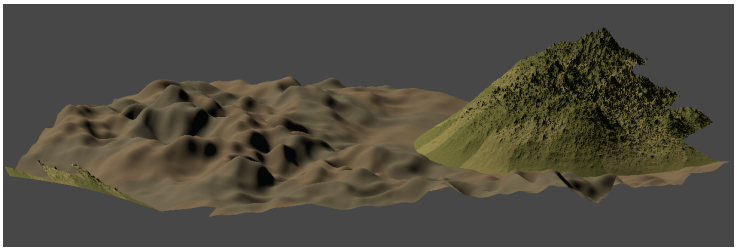
(a) Configuración de parámetros.



(b) Vista de pájaro con perímetros visibles.

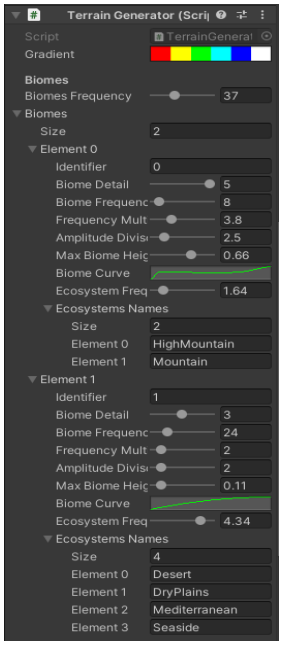


(c) Vista de pájaro.

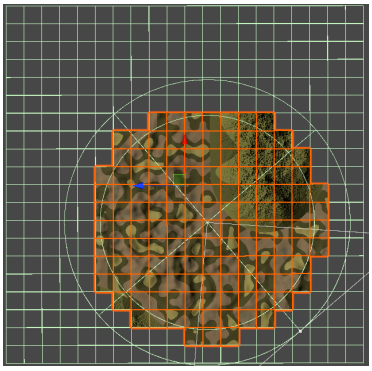


(d) Vista lateral.

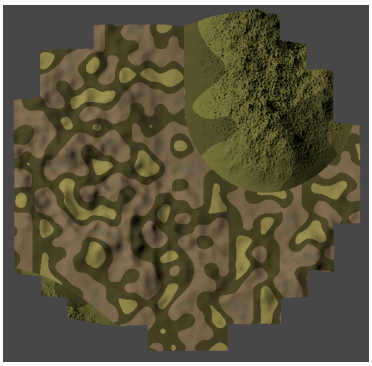
Figura 5.11: Resultados de la cuarta prueba



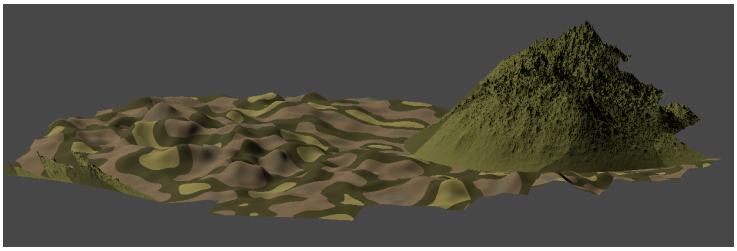
(a) Configuración de parámetros.



(b) Vista de pájaro con perímetros visibles.

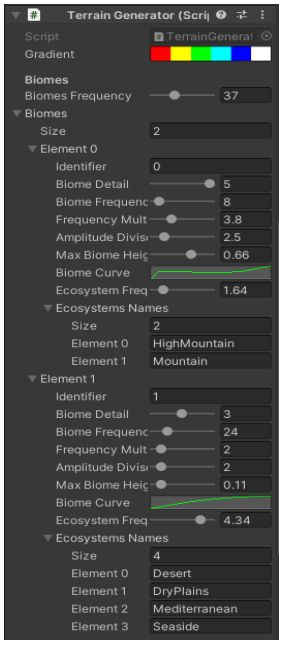


(c) Vista de pájaro.

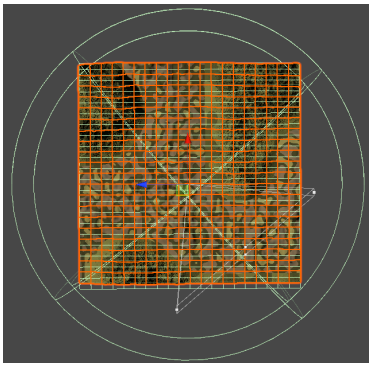


(d) Vista lateral.

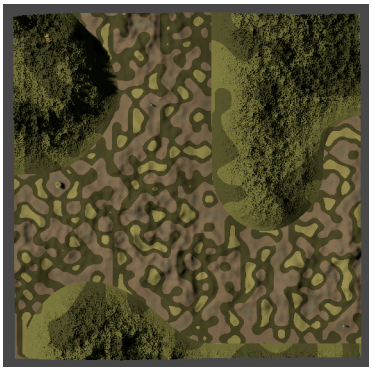
Figura 5.12: Resultados de la quinta prueba



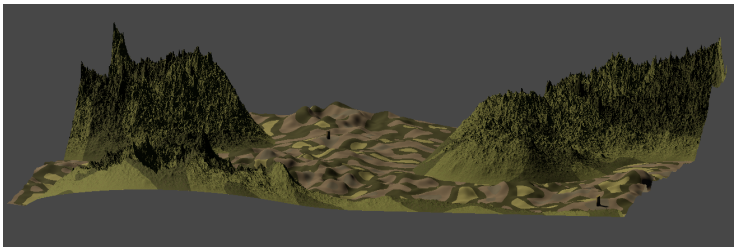
(a) Configuración de parámetros.



(b) Vista de pájaro con perímetros visibles.

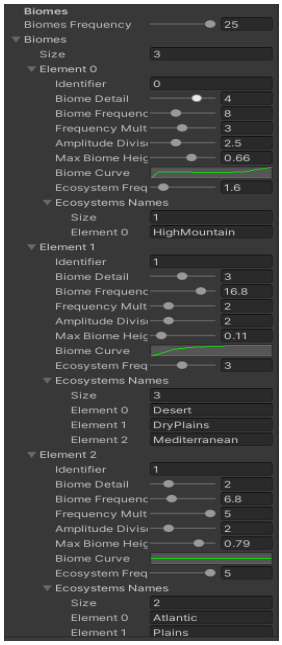


(c) Vista de pájaro.

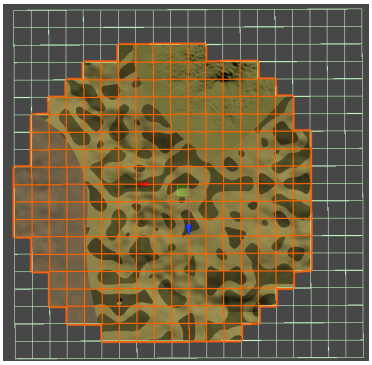


(d) Vista lateral.

Figura 5.13: Resultados de la sexta prueba

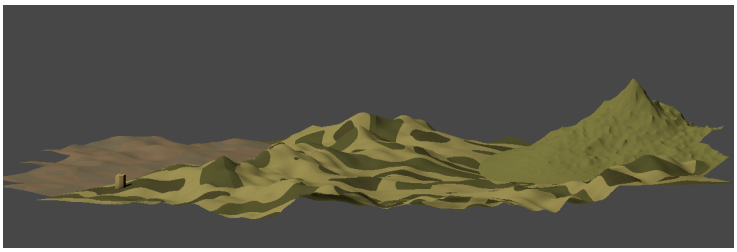


(a) Configuración de parámetros.



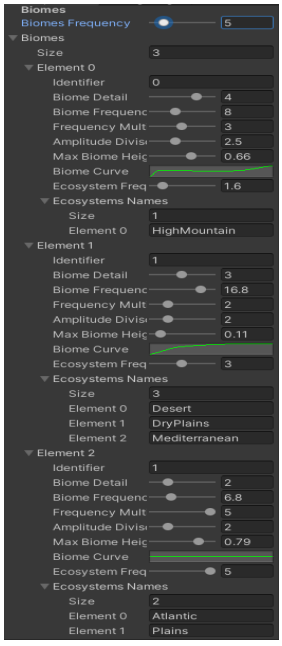
(b) Vista de pájaro con perímetros visibles.

(c) Vista de pájaro.

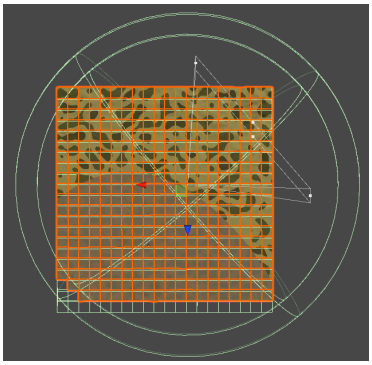


(d) Vista lateral.

Figura 5.14: Resultados de la séptima prueba

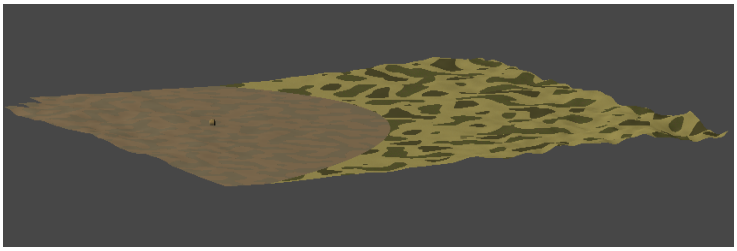


(a) Configuración de parámetros.



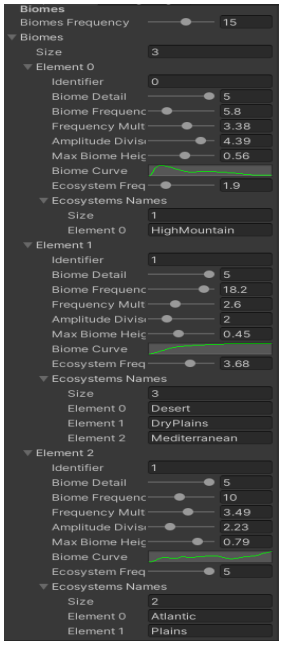
(b) Vista de pájaro con perímetros visibles.

(c) Vista de pájaro.

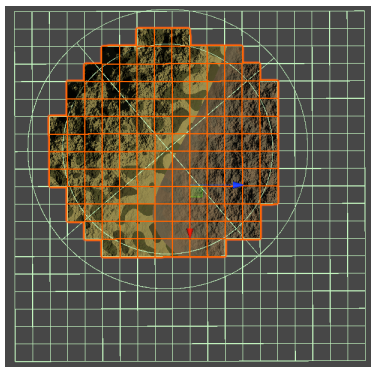


(d) Vista lateral.

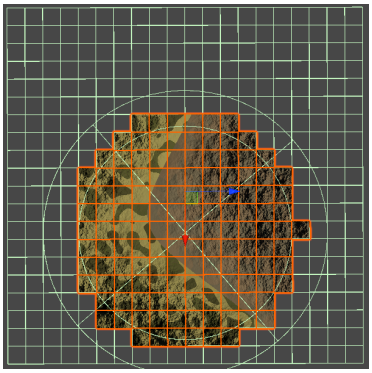
Figura 5.15: Resultados de la octava prueba



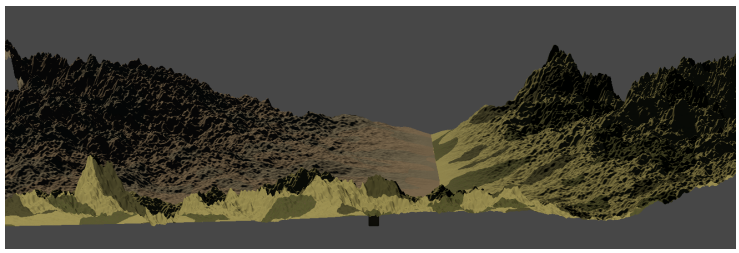
(a) Configuración de parámetros.



(b) Vista de pájaro con perímetros visibles.



(c) Vista de pájaro con perímetros visibles.



(d) Vista lateral.

Figura 5.16: Resultados de la novena prueba

En la séptima y octava prueba se obtiene un punto discordante en el terreno, es decir, que no se corresponde con el bioma circundante y provoca un salto brusco de altura. Se corregirá en la siguiente iteración.

En la novena prueba se observa el resultado de forzar los límites de frecuencia de los biomas. Provocando un terreno extremadamente irregular y poco realista, con lo que el cálculo de normales y tangentes de los vértices aumenta de forma considerable y, por consiguiente, el tiempo total de cálculo de la herramienta. Los límites serán redefinidos en la próxima iteración para asegurar un correcto funcionamiento.

Las ejecuciones varían entre 10 y 30 segundos (ver tabla 5.9), después de ese tiempo inicial la generación es en segundo plano e imperceptible. Dichas ejecuciones son dependientes del tamaño de los perímetros de cálculo y visualización y, en menor medida, del nivel de detalle de los biomas. El impacto en el rendimiento total del entorno 3D es entre bajo y medio dependiendo del nivel de detalle del entorno.

Prueba	Radio perímetro cálculo	Radio perímetro visualización	Tiempo inicial
1	1500 metros	1000 metros	10 segundos
2	1500 metros	1000 metros	12 segundos
3	1500 metros	1000 metros	12 segundos
4	2000 metros	1500 metros	15 segundos
5	2000 metros	1500 metros	15 segundos
6	3500 metros	3000 metros	30 segundos
7	2000 metros	1500 metros	15 segundos
8	3500 metros	3000 metros	25 segundos
9	2000 metros	1500 metros	20 segundos

Cuadro 5.9: Tiempo de cálculo inicial.

El resultado de la iteración en general es satisfactoria, se obtienen terrenos realistas y con un grado alto de personalización, al mismo tiempo se conserva la naturaleza automática de la herramienta, ya que solo se necesita una configuración previa. El sistema ya no se bloquea en el cálculo inicial, por lo que dicho cálculo puede ser ocultado tras una pantalla de carga.

5.2.3 Tercera Iteración

En esta iteración se aborda la generación de los elementos decorativos, la carga y su procesado, así como la correcta colocación en función de la pendiente del terreno. La generación se producirá en un conjunto de coordenadas resultado de un muestreo en disco de Poisson, dicho conjunto ya está disponible en el núcleo de la herramienta desde la primera iteración.

5.2.3.1 Análisis

Para la realización de esta iteración se modifica el módulo Terreno, que incorporará las nuevas funciones de generación y destrucción de decorados. A continuación se detallan las nuevas fases del proceso:

1. **Generación de decorados en base a muestreo:** Una vez publicados los datos del terreno para su renderizado se generan los decorados. Se itera a través de los valores de la muestra, que contienen coordenadas, ecosistema y elemento concreto, y se crean las entidades.

2. **Destrucción de decorados:** Una vez ocultados los datos del terreno se procede a la destrucción de las entidades asociadas al módulo Terreno.

5.2.3.1.1 Actores

Los actores intervinientes a lo largo de las iteraciones serán los mismos que en la primera iteración (véase sección 5.2.1.1.1).

5.2.3.1.2 Casos de uso

Se añade un nuevo caso de uso en esta iteración (ver figura A.7) detallado a continuación:

1. **Colocar decorados en carpeta de recursos:** El desarrollador coloca los decorados de los ecosistemas que desee generar en la carpeta de recursos. El nombre de dichos decorados ha de empezar por el nombre del ecosistema, para una correcta identificación durante el procesado y la generación. Este caso de uso se encuentra detallado en la tabla 5.10.

CU-6	**Colocar decorados en carpeta de recursos**
Descripción	El desarrollador coloca los decorados de los ecosistemas que desee generar en la carpeta de recursos y modifica sus nombres.
Precondición	Ninguna.
Poscondición	Todos los decorados están en la carpeta de recursos y con una nomenclatura correcta.
Actores	Desarrollador
Pasos	1. El desarrollador coloca cada elemento necesario en la carpeta de recursos. 2. El desarrollador modifica el nombre de los elementos para que empiecen por el nombre del ecosistema al que pertenecen, seguido del nombre identificativo del recurso (*Grass*, *Tree*, *Rock* o *Plant*).

Cuadro 5.10: CU-6: Colocar decorados en carpeta de recursos

5.2.3.2 Diseño

El diseño realizado durante la tercera iteración contiene las funcionalidades de generación y destrucción de decorados, así como la facilidad de añadir tantos decorados por ecosistema como se desee, siendo estos cargados desde una carpeta denominada *Resources*. En el diagrama de clases de la figura A.12, se ven por un lado, las variables de personalización añadidas a la clase *Biome*, y por otro lado, las variables referentes a decorados añadidas a la clase *TerrainGenerator*. Se explican a continuación dichas variables:

1. **grassThreshold, treesThreshold, rocksThreshold, plantsThreshold:** Son el umbral de aceptación del valor devuelto por el mapa de ruido de decorados. Si el valor sobrepasa

el umbral, las coordenadas de ese punto son añadidas al conjunto de elementos que se someterá al muestreo en disco de Poisson.

2. **grassDensity, treesDensity, rocksDensity, plantsDensity:** Son el nivel de densidad de elementos que se generarán en el terreno. A mayor densidad, mayor número de elementos por m^2 y mayor índice de solapamiento.

3. **Grass, Trees, Rocks, Plants:** Arrays con longitud igual al número de ecosistemas. Contienen listas dinámicas con los decorados de cada ecosistema.

4. **poissonGrassSamplings, poissonTreesSamplings, poissonRocksSamplings, poissonPlantsSamplings:** Listas de vectores resultado del muestreo en disco de Poisson, que contienen los parámetros de generación de decorados. Coordenadas, ecosistema y elemento concreto a generar.

Las clases *TerrainGenerator* y *MapGenerator* también pasan a guardar referencias a dos nuevas clases, *LoadResources* y *EntityGenerator*, que siguen el patrón de diseño *Singleton* [22]. Se opta por este patrón ya que se necesita un punto global de acceso a la información que poseen y sólo es necesario calcularla una vez. Dichas clases se pueden observar en la figura A.13 y serán explicadas a continuación:

- **LoadResources:** *Singleton* que inicializa la clase *MapGenerator*, pasándole el número de ecosistemas y la lista de nombres de los mismos. Esta clase busca en la carpeta de *Resources* todos los *GameObject* que empiecen por el nombre de cada uno de los ecosistemas, si el nombre continúa con la palabra clave que define el decorado (*Grass*, *Tree*, *Rock* o *Plant*), lo añade a la lista del tipo de decorado del array que le corresponde al ecosistema. Cuando finaliza su inicialización se queda a la espera como un punto de acceso global a los recursos.

- **EntityGenerator:** *Singleton* que inicializa la clase *MapGenerator*, pasándole el número de ecosistemas. Esta clase crea un tipo de diccionario denominado *NativeMultiHashMap*, necesario para usar el sistema *Jobs* de C#, ya que es una estructura *Thread safe* necesaria para hacer uso del *multithreading*. La clase genera un *NativeMultiHashMap* para cada tipo de decorado, almacenará como clave un int que representa el ecosistema y como valor una *Entity*. Para generar los *NativeMultiHashMap* se itera sobre los arrays de listas que ofrece *LoadResources*, donde cada índice del array ofrece el identificador del ecosistema y todos los *GameObject* de la lista se transforman en *Entities*.

Para la generación y destrucción de decorados se necesitan otras dos clases, *GenerateInstancesJob* y *EntityDestroyerSystem*. Para identificar qué elementos destruir se crea una tercera clase, *IdentifierComponent*, que implementa el interfaz *IComponent* y define una variable que

identificará el fragmento de terreno, dicho componente será asignado a cada *Entity* para posteriormente filtrarlas. A continuación se detallan las clases *GenerateInstancesJob* y *EntityDestroyerSystem*:

- **GenerateInstancesJob:** Estructura que implementa el interfaz *IJob* y que contiene su funcionalidad en el método *Execute()*. Cuando la clase *TerrainGenerator* recibe la llamada al método *InRangeVisualization* ejecuta el método *GenerateDecoration*, que crea una nueva estructura de tipo *GenerateInstancesJob*, con los datos que posee y los datos referentes a *Entities* que solicita al *Singleton EntityGenerator*. Tras generar la estructura llama al método *Execute()*. Este método itera sobre las listas de elementos resultado del muestreo en disco de Poisson, instancia todas las *Entities* adecuándolas a la pendiente del terreno y les asigna el componente *IdentifierComponent* con el valor del identificador del fragmento de terreno.

- **EntityDestroyerSystem:** Clase que hereda de *ComponentSystem*. Cuando la clase *TerrainGenerator* recibe la llamada al método *OutRangeVisualization* crea una instancia de esta clase y ejecuta el método *DestroyEntities* pasándole su identificador como argumento. Este método itera en paralelo sobre todas las *Entities*, que existen en el mundo definido por el motor de videojuego, y destruye aquellas cuyo *IdentifierComponent* sea igual al que se le pasa como argumento. De esta manera, cada fragmento de terreno solo destruirá las *Entities* que están en su interior.

Debido a que cada *TerrainGenerator* está aislado y no se conocen entre ellos, será la clase *MapGenerator* la que asigne los identificadores.

5.2.3.3 Implementación

Durante la implementación de esta iteración se corrigió un error de codificación que provocaba los saltos bruscos de terreno detectados durante las pruebas de la iteración anterior. Dicho error se producía durante la primera evaluación del mapa de ruido, la que determina en qué zona se genera cada bioma. Debido a que se usaban operaciones ”mayor que” y ”menor que” excluyentes, se dejaba el punto intermedio evaluado al bioma por defecto (con índice 0), lo que provocaba que algunas veces no coincidiera dicho punto con el bioma circundante, y de ahí que tuviese altura, perfil y textura diferente.

Se crearán cinco *scripts*, uno para cada nueva clase. No es necesaria la creación de ningún *GameObject*. Se crea una jerarquía de carpetas dentro de Resources para albergar los *GameObject* que representan los decorados. Todos los decorados usados de ahora en adelante son descargados de la *Asset Store* de Unity, todos ellos gratuitos.

5.2.3.4 Pruebas

Todas las pruebas se realizan con misma semilla y el mismo equipo informático (véase tabla 5.4). Se variarán los perímetros de cálculo y visualización para comprobar el funcionamiento de la generación por distancia. Se mantendrá siempre el mismo rango de altura ([0, 2000]), ya que los biomas son los que definen sus diferentes alturas. Debido a la gran cantidad de parámetros referentes a biomas y ecosistemas, se detallaran los mismos en imágenes (véase sección 5.2.2.2 para definición de los mismos)

Se han realizado un total de 5 pruebas, detalladas en la tabla 5.11.

Cuadro 5.11: Pruebas realizadas en la tercera iteración

Nº	Descripción	Resultado esperado	Resultado obtenido
1	Generación de un entorno con los parámetros referidos a biomas detallados en la figura 5.17a. 1000 metros de radio de visualización y 1500 metros de radio de cálculo.	Terreno compuesto de 1 bioma y 2 ecosistemas, baja densidad de cambios de altura y nivel de detalle bajo. Solo se generan los fragmentos dentro del perímetro de visualización. Terreno continuo sin cambios bruscos de altura. Densidad media de elementos de decoración.	Terreno compuesto de 1 bioma y 2 ecosistemas, baja densidad de cambios de altura y nivel de detalle bajo. Solo se generan los fragmentos dentro del perímetro de visualización. Terreno continuo sin cambios bruscos de altura. Densidad media de elementos de decoración. (Ver figura 5.17)
2	Generación de un entorno con los parámetros referidos a biomas detallados en la figura 5.18a. 1000 metros de radio de visualización y 1500 metros de radio de cálculo.	Terreno compuesto de 1 bioma y 2 ecosistemas, baja densidad de cambios de altura y nivel de detalle bajo. Solo se generan los fragmentos dentro del perímetro de visualización. Terreno continuo sin cambios bruscos de altura. Densidad muy alta de elementos de decoración.	Terreno compuesto de 1 bioma y 2 ecosistemas, baja densidad de cambios de altura y nivel de detalle bajo. Solo se generan los fragmentos dentro del perímetro de visualización. Terreno continuo sin cambios bruscos de altura. Densidad muy alta de elementos de decoración. (Ver figura 5.18)

Cuadro 5.11: Pruebas realizadas en la tercera iteración (continuación)

Nº	Descripción	Resultado esperado	Resultado obtenido
3	Generación de un entorno con los parámetros referidos a biomas detallados en la figura 5.19a. 1000 metros de radio de visualización y 1500 metros de radio de cálculo.	Terreno compuesto de 1 bioma y 2 ecosistemas, baja densidad de cambios de altura y nivel de detalle bajo. Solo se generan los fragmentos dentro del perímetro de visualización. Terreno continuo sin cambios bruscos de altura. Densidad alta de elementos de decoración.	Terreno compuesto de 1 bioma y 2 ecosistemas, baja densidad de cambios de altura y nivel de detalle bajo. Solo se generan los fragmentos dentro del perímetro de visualización. Terreno continuo sin cambios bruscos de altura. Densidad alta de elementos de decoración. (Ver figura 5.19)
4	Generación de un entorno con los parámetros referidos a biomas detallados en la figura 5.20b. 2000 metros de radio de visualización y 2500 metros de radio de cálculo.	Terreno compuesto de 2 biomas y 4 ecosistemas, alta densidad de cambios de altura y nivel de detalle bajo. Solo se generan los fragmentos dentro del perímetro de visualización. Terreno continuo sin cambios bruscos de altura. Densidad media de elementos de decoración.	Terreno compuesto de 2 biomas y 4 ecosistemas, alta densidad de cambios de altura y nivel de detalle bajo. Solo se generan los fragmentos dentro del perímetro de visualización. Terreno continuo sin cambios bruscos de altura. Densidad media de elementos de decoración. (Ver figura 5.20)
5	Generación de un entorno con los parámetros referidos a biomas detallados en la figura 5.21b. 2000 metros de radio de visualización y 2500 metros de radio de cálculo.	Terreno compuesto de 2 biomas y 4 ecosistemas, densidad muy alta de cambios de altura y nivel de detalle alto. Solo se generan los fragmentos dentro del perímetro de visualización. Terreno continuo sin cambios bruscos de altura. Densidad alta de elementos de decoración.	Terreno compuesto de 2 biomas y 4 ecosistemas, densidad muy alta de cambios de altura y nivel de detalle alto. Solo se generan los fragmentos dentro del perímetro de visualización. Terreno continuo sin cambios bruscos de altura. Densidad alta de elementos de decoración. (Ver figura 5.21)

En todas las pruebas se observa cómo, debido al muestreo en disco de Poisson, el sistema

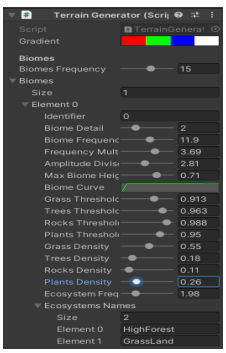
genera los decorados de manera uniforme y no solapante, al mismo tiempo que conserva un aspecto caótico natural. Gracias al sistema de instanciación y renderizado en *multithreading* de las *Entities*, los decorados se generan casi instantáneamente, algo imposible con *GameObjects*. Por el diseño ligero orientado a los datos y la agrupación de memoria en *chuncks*, se pueden llegar a generar 100 *Entities* en el mismo tiempo que un *GameObject*, siendo cada *Entity* una copia del *GameObject*, es decir, un aumento del 10.000% en cuanto a eficiencia.

Dicha eficiencia se comprueba en la segunda prueba, donde tenemos una densidad de decorados muy alta con umbrales muy bajos, llegando a generarse en torno a 80.000 *Entities* en menos de 10 km^2 (el perímetro de visualización no ocupa todo el mapa). Para poner estos datos en contexto, 50.000 es el número máximo de *GameObjects* que Unity puede manejar en una escena, tras lo cual habría que implementar alguna técnica de combinación de modelos 3D, con lo que se perdería la unicidad de los elementos. Aun con la cantidad de elementos que se generan no se pierde una gran cantidad de rendimiento, aproximadamente el 50% de carga en la GPU y 15% de carga en la CPU (ver tabla 5.4 para detalles del equipo informático).

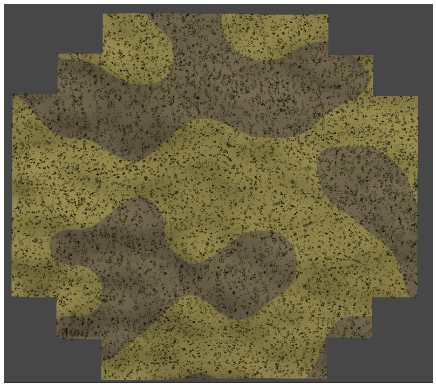
En la tercera prueba se puede observar la mezcla de ecosistemas implementada en los bordes de los mismos, mezclándolos para mayor realismo. Esta técnica es peligrosa si, por ejemplo, colocamos dos ecosistemas muy diferentes dentro del mismo bioma, como el bosque y la montaña nevada que se pueden observar en la figura 5.19. La incompatibilidad de estos biomas provoca una pérdida de naturalidad del entorno. En la figura 5.20 se observa que, al colocar dos ecosistemas montañosos en un bioma y dos ecosistemas boscosos en otro, se genera la naturalidad deseada y transiciones más suaves.

En la quinta prueba se pone a prueba el sistema de adaptación al terreno de los elementos, generando un terreno muy abrupto, con mucho detalle y grandes desniveles. En la figura 5.21 se observa como en la mayoría de las situaciones los decorados se adaptan a cualquier desnivel que no sea considerado vertical (máximo 70º de inclinación), pero termina por fallar cuando el decorado tiene mucha extensión y el desnivel muy poca, quedando el decorado flotando sobre la superficie que tendría el terreno si éste continuase con la misma pendiente.

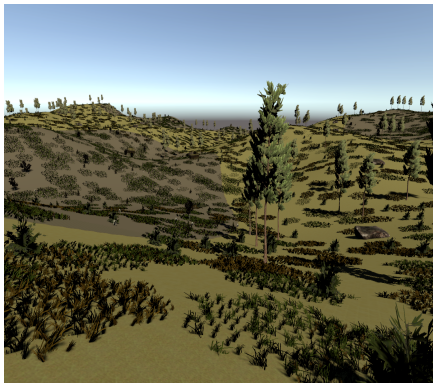
Para solucionar este problema se implementará en la siguiente iteración un sistema más complejo que asegure la correcta colocación de estos decorados.



(a) Configuración de parámetros.

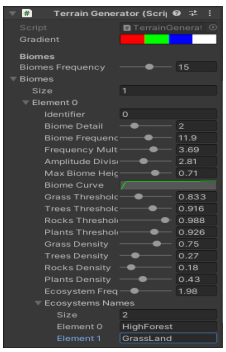


(b) Vista aérea

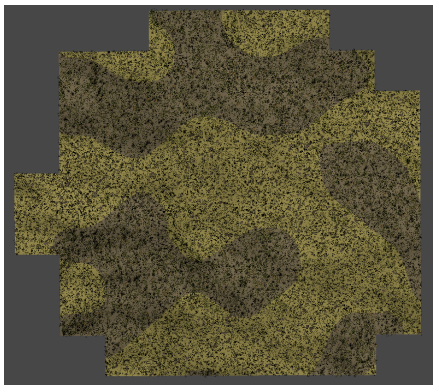


(c) Vista detallada en primera persona.

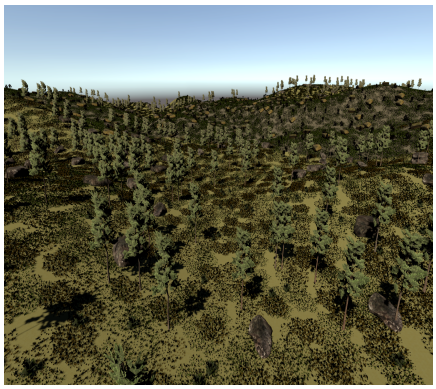
Figura 5.17: Resultados de la primera prueba



(a) Configuración de parámetros.

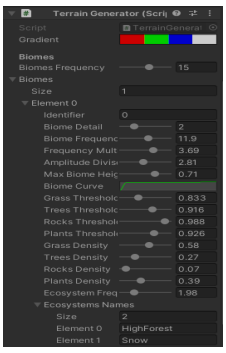


(b) Vista aérea

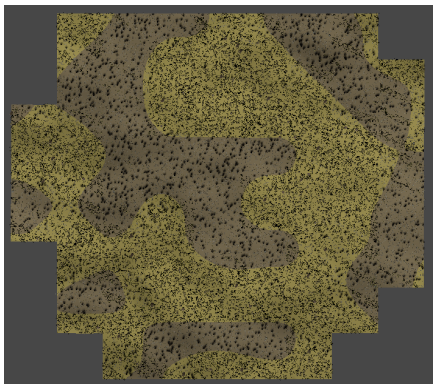


(c) Vista detallada en primera persona.

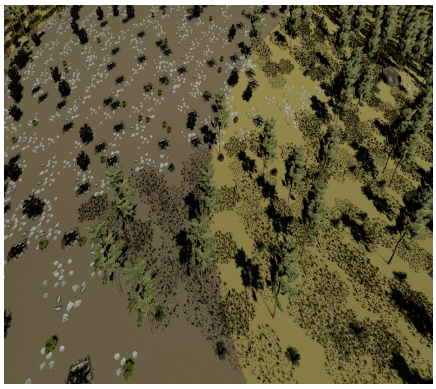
Figura 5.18: Resultados de la segunda prueba



(a) Configuración de parámetros.

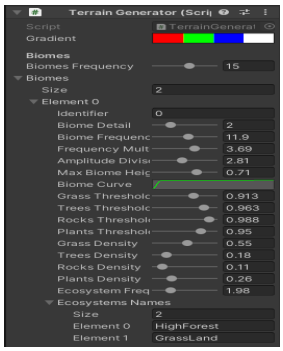


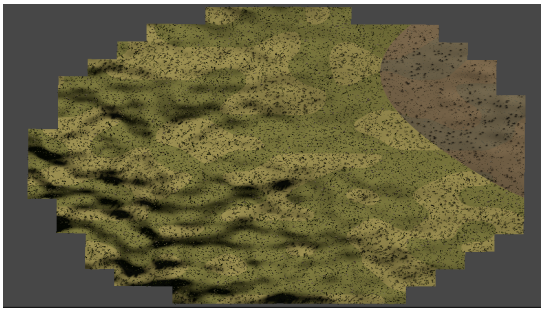
(b) Vista aérea



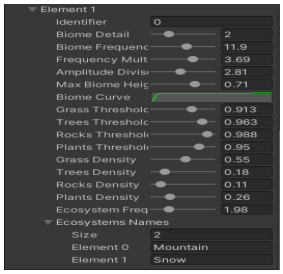
(c) Vista detallada en primera persona.

Figura 5.19: Resultados de la tercera prueba

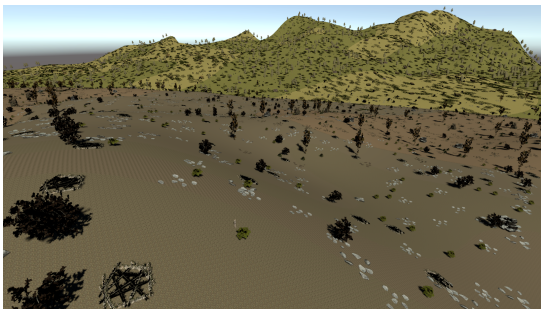




(a) Vista aérea

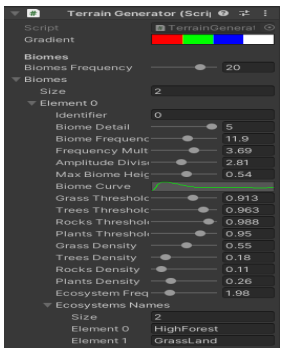


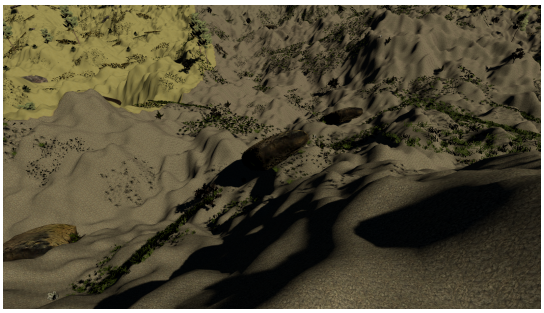
(b) Configuración de parámetros.



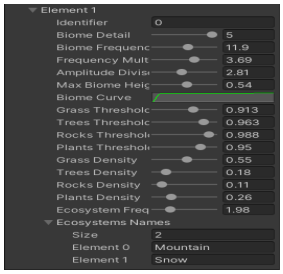
(c) Vista detallada en primera persona.

Figura 5.20: Resultados de la cuarta prueba

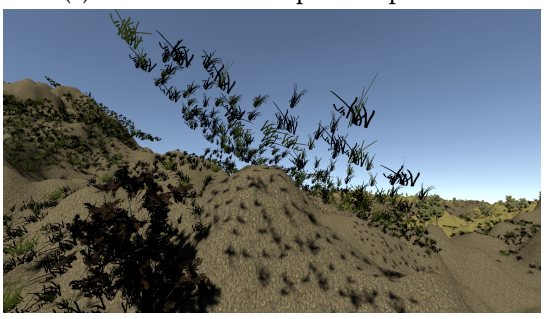




(a) Vista detallada en primera persona.



(b) Configuración de parámetros.



(c) Vista detallada del problema del tamaño de los decorados.

Figura 5.21: Resultados de la quinta prueba

Los tiempos de ejecución son similares a la iteración anterior (ver tabla 5.9), con una desviación de menos de 3 segundos debido a la alta eficiencia de generación y renderizado de las *Entities*.

El resultado de la iteración en general es satisfactoria, se obtienen terrenos superpoblados de elementos decorativos lo que aumenta mucho su nivel de realismo. El impacto en el sistema aumenta, llegando a alcanzar el límite si se pone el nivel de detalle al máximo, con densidad máxima de decorados y perímetros de cálculo y visualización con radio superior a 4 km. Se pierde la barrera de la continuidad (30 fotogramas por segundo) en torno a las 300.000 *Entities*.

5.2.4 Cuarta Iteración

En esta iteración se plantea la creación de un sistema que permita aprovechar el rendimiento de las *Entities* al tiempo que interactúan con el mundo original de Unity.

También se implementará el agua y un semáforo, que controla las funciones de cálculo y visualización de cada fragmento de terreno, para que se ejecuten en el orden correcto y no se solapen.

5.2.4.1 Análisis

Para que un objeto dentro de un mundo 3D sea interactuable, debe poseer las cualidades físicas que tendría en el mundo real. En el caso concreto de este proyecto, los decorados como rocas, árboles o arbustos deben ser sólidos, tener masa y ser inamovibles hasta un cierto umbral de fuerza aplicada. Los cálculos de estas físicas sobre todos los decorados consumiría demasiados recursos del sistema, por lo que se opta por simplificarlo en dos principios: Los objetos son sólidos e inamovibles.

Con un *Collider* con la forma del objeto 3D basta para emular solidez e inamovilidad. El problema viene dado por que los decorados son *Entities*, que a pesar de tener sus propios *colliders* no se pueden comunicar con los objetos convencionales, donde normalmente se crea la lógica del videojuego o simulador. Esto es debido a que usan motores de físicas distintos, todo el *Entity Component System* usa el motor Havok y todos los *GameObjects* usan el motor Physx.

Para que estos dos motores de físicas se comuniquen entre sí, se pueden elaborar complejos métodos adaptadores que traduzcan las llamadas de un motor a otro. Alternativamente y debido a la gran complejidad de estos sistemas, se usará un método detector muy utilizado en la comunidad de Unity.

En dicho método se usa un objeto que, varias veces por segundo, genera un *Collider* del motor Havok en la posición de la cámara. Este *Collider* detectará si hay alguna *Entity* dentro de su perímetro y dará acceso a sus propiedades, tras lo cual se podrá replicar como *GameObject* y añadirle un *Collider* que reproduzca las características requeridas.

Para implementar el agua solo será necesario un modelo de agua, con dimensiones 1x1, que se pueda escalar al tamaño del fragmento de terreno que lo aloje. Un umbral determinará a qué nivel por encima de la altura mínima se encontrará el agua.

5.2.4.1.1 Actores

Los actores intervinientes a lo largo de las iteraciones serán los mismos que en la primera iteración (véase sección 5.2.1.1.1).

5.2.4.1.2 Casos de uso

Se añade un nuevo caso de uso durante esta iteración (ver figura A.8).

1. **Añadir componente a decorados que requieran *Collider*:** El desarrollador añade, a los decorados que requieran un *Collider*, un componente para identificarlos posteriormente. Este caso de uso se encuentra detallado en la tabla 5.10.

CU-7	**Añadir componente a decorados que requieran colisionador**
Descripción	El desarrollador añade, a los decorados que requieran un colisionador, un componente para identificarlos posteriormente.
Precondición	Los decorados han de estar en la carpeta de recursos.
Poscondición	Todos los decorados que requieran un *Collider* tienen el componente añadido.
Actores	Desarrollador
Pasos	1. El desarrollador selecciona todos los decorados que requieran un *Collider*. 2. El desarrollador añade a estos decorados el componente identificador.

Cuadro 5.12: CU-7: Añadir componente a decorados

5.2.4.2 Diseño

El diseño realizado durante la cuarta iteración puede observarse en la figura A.14. Se añade un nuevo sistema, compuesto de dos clases, que se encargará de generar los *Collider*, un componente que será añadido a los decorados y nuevas variables a la clase *TerrainGenerator* necesarias para el semáforo.

El componente que será añadido a los decorados se define en la clase *DecorationComponent* que implementa el interfaz *IComponentData*. Esta clase está vacía, pues el hecho de poseer el componente ya funciona como identificador.

El nuevo sistema de generación de *Colliders* se compone de una clase principal, *CollisionECS*, y una auxiliar, *RemoteDestroy*, explicadas en detalle a continuación:

1. ***CollisionECS:*** Esta clase hereda de *Monobehaviour* e implementa su funcionamiento en el método heredado *Update*, que se ejecuta en cada ciclo de renderizado, comprobando si el tiempo para la siguiente detección ha transcurrido y, de ser el caso, llamando al método *SphereCast*. Sus variables y métodos principales se explican a continuación:

    - **template:** *GameObject* plantilla que se usará para instanciar los *Collider* y que implementan la clase *RemoteDestroy*.
    - **createRadius:** Radio del perímetro de detección.
    - **entityCollided:** *Entity* que representa una de las entidades que ha entrado dentro del perímetro de detección.
    - **physicsWorldSystem:** Sistema que representa el mundo físico de las *Entities*, implementado con Havok.
    - **collisionWorld:** Mundo que gestiona las colisiones entre *Entities*, implementado en Havok.
    - **list:** Lista de coordenadas de las *Entities* que entran dentro del perímetro de detección. Controla que no se generen dos *Collider* para la misma *Entity*.
    - **manager:** Sistema que nos permite iterar sobre las *Entities* que poseen un componente de tipo *DecorationComponent* y acceder a sus propiedades.
    - **spherecastTime:** Tiempo de espera entre detecciones.
    - **SphereCast:** Método que proyecta una esfera de radio *createRadius* en el mundo físico de las *Entities*. Si esta esfera colisiona con alguna *Entity*, se comprueba la lista para asegurarse de que no se haya detectado antes. Si no se ha detectado previamente, se instancia un *GameObject* de tipo *template* y se le otorga la posición y el modelo 3D de la *Entity*, tras lo cual, se llama al método *SetCollisionECS* del componente *RemoteDestroy*, pasándose a sí mismo como argumento.
    - **RemoveEntity:** Método que elimina un identificador de la lista de identificadores de *Entities*.

2. ***RemoteDestroy:*** Clase que implementan todos los *GameObject* que instancia *CollisionECS*. Se autodestruye pasado un tiempo para liberar recursos. Sus variables y métodos principales se explican a continuación:

    - **TTL:** Tiempo de vida en segundos que tiene el *GameObject*. Se reduce en cada ciclo de renderizado en el método heredado *Update*.
    - **referencedCollisionECS:** Clase *CollisionECS* que ha instanciado este objeto. Justo antes de que el objeto actual se autodestruya, llama al método *RemoveEntity* de esta clase para que elimine su identificador.

- **SetCollisionECS:** Método que sirve para guardar referencia a la clase *CollisionECS* que ha instanciado el objeto actual.

A grandes rasgos tenemos un sistema pegado a la cámara que visualiza el entorno 3D, que detecta las *Entities* dentro de su perímetro y genera *GameObjects* con *Colliders* que simulan dichas *Entities*. Estos *GameObjects* se autodestruyen pasado un tiempo, por lo que no llegan a colapsar el motor de videojuego. Justo antes de autodestruirse liberan el identificador de la *Entity* para que pueda volver a ser detectada y se pueda volver a instanciar un nuevo *GameObject* que la represente.

Para la generación del agua simplemente se comprobará que la altura de cada vértice del fragmento del terreno no supere el umbral del agua (*waterHeight*). Si lo supera se instancia el modelo de agua (*GameObject water*) en las coordenadas del fragmento de terreno, con altura igual al umbral dado y dimensiones iguales a las del fragmento de terreno.

Para la implementación del semáforo se definen 8 nuevas variables, cuatro *bool*, que garanticen el correcto orden de ejecución, y cuatro *Coroutine*, que identifiquen los 4 métodos que se ejecutan de forma asíncrona. A continuación se detalla su funcionamiento:

- **outCalculus:** Se evalúa a *false* cuando el método *OutRangeCalculus* inicia su ejecución. Cuando el método finaliza se evalúa a *true*.

- **outVisualization:** Se evalúa a *false* cuando el método *OutRangeVisualization* inicia su ejecución. Cuando el método finaliza se evalúa a *true*.

- **soilCalculusDone:** Se evalúa a false cuando se inicia el método asíncrono *CalculateBackground*. Cuando el método finaliza se evalúa a *true*. Si el método asíncrono *GeneratePoissonDist* se inicia siendo esta variable *false*, para su ejecución hasta que sea *true*.

- **poissonCalculusDone:** Se evalúa a false cuando se inicia el método asíncrono *GeneratePoissonDist*. Cuando el método finaliza se evalúa a *true*.

- **inSoilCalculus:** Corrutina que representa el método asíncrono *CalculateBackground*. El método *InRangeCalculus* ejecuta este método si es *null*. El método *OutRangeCalculus* detiene este método y lo evalúa a *null*.

- **inDecorationCalculus:** Corrutina que representa el método asíncrono *GeneratePoissonDist*. El método *InRangeCalculus* ejecuta este método si es *null*. El método *OutRangeCalculus* detiene este método y lo evalúa a *null*.

- **inSoilVisualization:** Corrutina que representa el método asíncrono *EnableVisualization*. El método *InRangeVisualization* ejecuta este método si es *null*. El método *OutRangeVisualization* detiene este método y lo evalúa a *null*.

- **inDecorationVisualization:** Corrutina que representa el método asíncrono *GenerateDecoration*. El método *InRangeVisualization* ejecuta este método si es *null*. El método *OutRangeVisualization* detiene este método y lo evalúa a *null*.

A grandes rasgos, los métodos *OutRangeCalculus* y *OutRangeVisualization* detienen los métodos asíncronos correspondientes. Dichos métodos se quedan en espera si se vuelven a ejecutar antes de que las variables *outCalculus* o *outVisualization* se evalúen a *true*. El método asíncrono *GeneratePoissonDist* es dependiente de la finalización del método *CalculateBackground*. El método *EnableVisualization* es dependiente de la finalización del método *CalculateBackground*, si el último se ejecutan antes de que primero finalice, se queda a la espera de que la variable *soilCalculusDone* se evalúe a true. El método *GenerateDecoration* es dependiente de la finalización del método *GeneratePoissonDist*, si el último se ejecutan antes de que primero finalice, se queda a la espera de que la variable *poissonCalculusDone* se evalúe a true.

5.2.4.3 Implementación

Para la implementación se crean 3 *scripts* que albergan las clases *DecorationComponent*, *CollisionECS* y *RemoteDestroy*.

Se crea un nuevo *GameObject* que tiene como componente el *script CollisionECS* y se coloca como hijo en la jerarquía de la cámara que visualiza el entorno, de esta manera se desplaza y rota en las misma direcciones que la cámara, por lo tanto, el perímetro de detección de *Entities* siempre estará junto a la cámara. Se crea un *GameObject* plantilla que contiene el *script RemoteDestroy* y se asigna al *script CollisionECS* para que pueda crear instancias del mismo cuando lo necesite.

Se añade el *script DecorationComponent* a todos los *GameObjects* que necesiten ser sólidos. Cuando sean transformados en *Entities* por la clase *EntityGenerator* conservaran el componente que los identificara como *Colliders* a crear.

Se modifica el *script TerrainGenerator* para añadir las nuevas variables. También se añade un *GameObject* gratuito sacado de la *Asset Store* que representa el agua.

5.2.4.4 Pruebas

Todas las pruebas se realizan con misma semilla y el mismo equipo informático (véase tabla 5.4). Se mantendrá siempre el mismo rango de altura ([0, 2000]), mismos perímetros de cálculo y visualización y tamaño de entorno de 5 km^2. Se toman siempre los mismos parámetros referentes a biomas y ecosistemas, detallados en la figura 5.22, ya que la única diferencia estaría en la altura máxima, que pondría a cada bioma por encima o por debajo del nivel del agua. Los restantes parámetros no tendrán influencia en las pruebas de las funcionalidades desarrolladas en esta iteración. (Véase sección 5.2.2.2 para definición de los mismos).

Se han realizado un total de 5 pruebas, detalladas en la tabla 5.13.

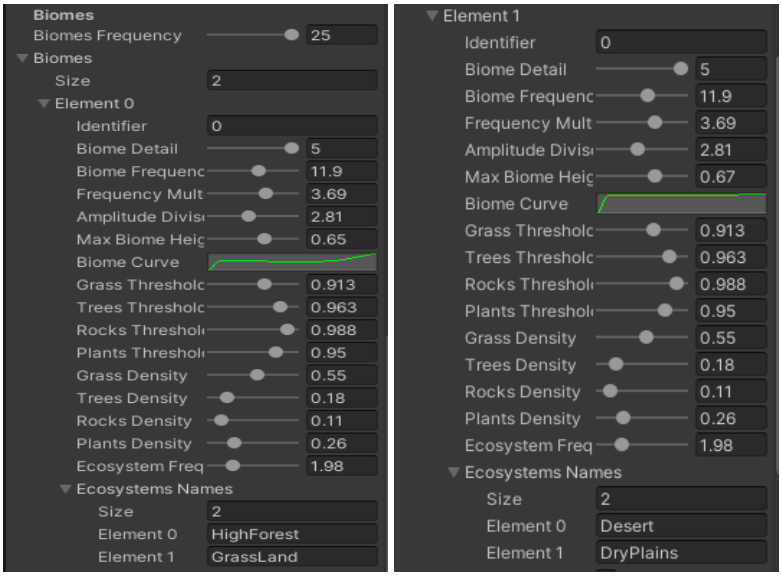


Figura 5.22: Configuración de parámetros.

Cuadro 5.13: Pruebas realizadas en la cuarta iteración

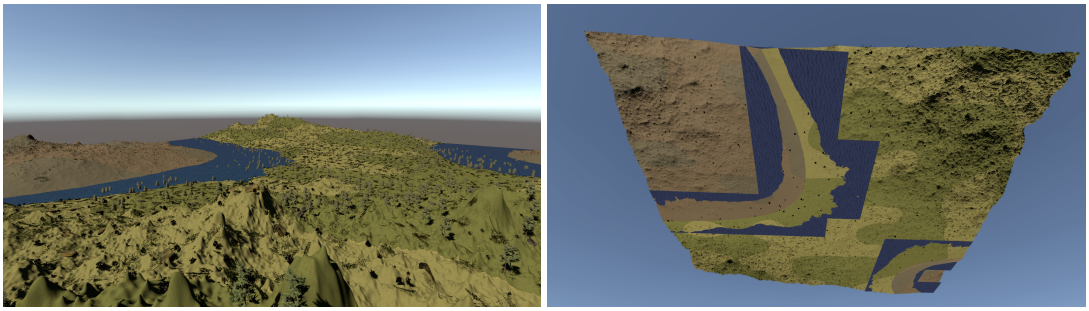
Nº	Descripción	Resultado esperado	Resultado obtenido
1	Generación de un entorno con los parámetros referidos a biomas detallados en la figura 5.22. 3000 metros de radio de visualización y 3500 metros de radio de cálculo.	Terreno compuesto de 2 biomas y 4 ecosistemas, alta densidad de cambios de altura y nivel de detalle alto. Solo se generan los fragmentos dentro del perímetro de visualización. Terreno continuo sin cambios bruscos de altura. Densidad media de elementos de decoración. Nivel del mar 15 metros sobre la altura mínima del terreno.	Terreno compuesto de 2 biomas y 4 ecosistemas, alta densidad de cambios de altura y nivel de detalle alto. Solo se generan los fragmentos dentro del perímetro de visualización. Terreno continuo sin cambios bruscos de altura. Densidad media de elementos de decoración. Nivel del mar 15 metros sobre la altura mínima del terreno. (Ver figura 5.23)

Cuadro 5.13: Pruebas realizadas en la cuarta iteración (continuación)

Nº	**Descripción**	**Resultado esperado**	**Resultado obtenido**
2	Generación de un entorno con los parámetros referidos a biomas detallados en la figura 5.22. 3000 metros de radio de visualización y 3500 metros de radio de cálculo.	Terreno compuesto de 2 biomas y 4 ecosistemas, alta densidad de cambios de altura y nivel de detalle alto. Solo se generan los fragmentos dentro del perímetro de visualización. Terreno continuo sin cambios bruscos de altura. Densidad media de elementos de decoración. Nivel del mar 50 metros sobre la altura mínima del terreno.	Terreno compuesto de 2 biomas y 4 ecosistemas, alta densidad de cambios de altura y nivel de detalle alto. Solo se generan los fragmentos dentro del perímetro de visualización. Terreno continuo sin cambios bruscos de altura. Densidad media de elementos de decoración. Nivel del mar 50 metros sobre la altura mínima del terreno. (Ver figura 5.24)
3	Generación de un *collider* al aproximarse a menos de 10 metros de una *Entity*	A más de 10 metros de distancia de cualquier *Entity* no se generará ningún *Collider* y dicha *Entity* no será interactuable. Tras acercar la cámara a menos de 10 metros de cualquier *Entity* se generará un *Collider* con su forma y en su posición exacta.	A más de 10 metros de distancia de cualquier *Entity* no se generará ningún *Collider* y dicha *Entity* no será interactuable. Tras acercar la cámara a menos de 10 metros de cualquier *Entity* se generará un *Collider* con su forma y en su posición exacta. (Ver figura 5.26)
4	Generación de un *collider* al aproximarse a menos de 15 metros de una *Entity*	A más de 15 metros de distancia de cualquier *Entity* no se generará ningún *Collider* y dicha *Entity* no será interactuable. Tras acercar la cámara a menos de 15 metros de cualquier *Entity* se generará un *Collider* con su forma y en su posición exacta.	A más de 15 metros de distancia de cualquier *Entity* no se generará ningún *Collider* y dicha *Entity* no será interactuable. Tras acercar la cámara a menos de 15 metros de cualquier *Entity* se generará un *Collider* con su forma y en su posición exacta. (Ver figura 5.27)

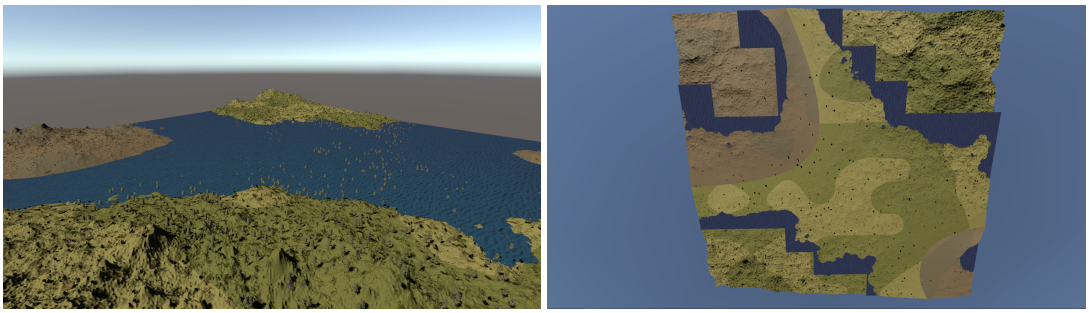
Cuadro 5.13: Pruebas realizadas en la cuarta iteración (continuación)

Nº	Descripción	Resultado esperado	Resultado obtenido
5	Generación de varios *colliders* al aproximarse a menos de 10 metros de varias *Entities*	A más de 10 metros de distancia de cualquier *Entity* no se generará ningún *Collider* y dicha *Entity* no será interactuable. Tras acercar la cámara a menos de 10 metros de varias *Entities* se generará un *Collider* por cada una con su forma y en su posición exacta.	A más de 10 metros de distancia de cualquier *Entity* no se generará ningún *Collider* y dicha *Entity* no será interactuable. Tras acercar la cámara a menos de 10 metros de varias *Entities* se generará un *Collider* por cada una con su forma y en su posición exacta. (Ver figura 5.28)



(a) Vista aérea (b) Vista bajo tierra.

Figura 5.23: Resultados de la primera prueba



(a) Vista aérea (b) Vista bajo tierra.

Figura 5.24: Resultados de la segunda prueba

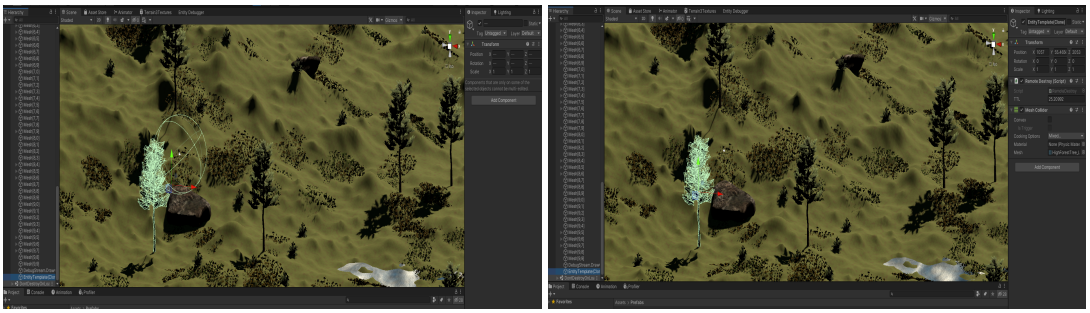
En las dos primeras pruebas se puede observar cómo cada fragmento de terreno, si contiene algún punto debajo del nivel del mar, crea un fragmento de agua. El fragmento de agua

se interseca con el terreno para formar costas, dando un nivel de realismo alto. Si todos los puntos del fragmento de terreno están por encima del nivel del mar no se crea fragmento de agua, como se puede ver en las imágenes subterráneas.

Figura 5.25: Representación del perímetro de detección sin tocar ninguna *Entity*.

En la figura 5.25 se observa la representación del perímetro de detección que genera la clase *CollisionECS* en el mundo físico de las *Entities*, así como la ausencia de *Colliders*, pues el perímetro no toca ninguna *Entity*. Este perímetro se usa en las pruebas 3, 4 y 5.

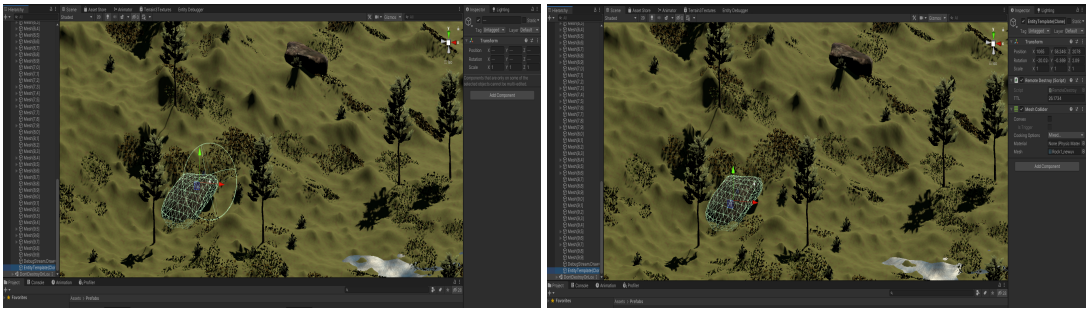
En la tercera y cuarta prueba se puede observar la creación de un *GameObject* con *Collider* cuando el perímetro entra en colisión con una *Entity*. Dicho *Collider* tiene la forma del modelo 3D de la *Entity* y un TTL descendente que, al llegar a cero, destruirá el objeto.



(a) Representación del perímetro de detección y creación del *Collider*.

(b) Vista detallada del *Collider* y el TTL que determinará su destrucción.

Figura 5.26: Resultados de la tercera prueba



(a) Representación del perímetro de detección y creación del *Collider*.

(b) Vista detallada del *Collider* y el TTL que determinará su destrucción.

Figura 5.27: Resultados de la cuarta prueba

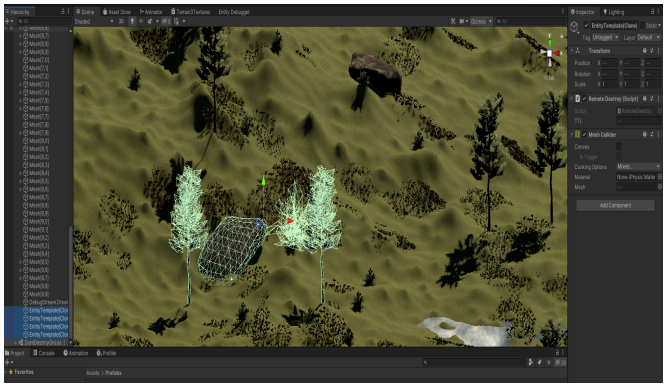


Figura 5.28: Resultados de la quinta prueba.

En la quinta prueba se puede observar cómo se crean múltiples *Colliders* a medida que las *Entities* entran en el perímetro de detección.

Los tiempos de ejecución son iguales a la iteración anterior, la creación de *Colliders* no genera carga computacional relevante. Para que dicha creación tuviese impacto, el radio de detección tendría que ser de cientos de metros y el entorno albergar decenas de miles de *Entities*.

El resultado de la iteración en general es satisfactoria, con el añadido del agua aumenta mucho el realismo y con la generación de *Colliders* los decorados pasan a ser interactuables.

5.2.5 Quinta Iteración

Durante esta iteración se aborda la optimización de los muestreos en disco de Poisson y la inclusión de construcciones prefabricadas que aporten civilización al entorno natural.

5.2.5.1 Análisis

Para la realización de esta iteración se modifica el módulo Entorno, creado en la primera iteración y no modificado desde entonces, para que tome el control de qué módulo Terreno aloja cada construcción prefabricada.

Como el módulo Terreno hace sus cálculos bajo demanda, y no en el momento de su creación, se debe habilitar en su interior una comprobación de características, como la pendiente o la altura sobre el nivel del mar, que determine si es apropiado para alojar una construcción.

El funcionamiento e interacción entre los módulos se observa en el diagrama de flujo de la figura A.4. A continuación se detallan las nuevas fases del proceso:

1. **Seleccionar fragmento para alojar prefabricado:** Tras la inicialización de los módulos Terreno, se seleccionan los que van alojar los prefabricados. Como todavía no se tienen detalles de cómo va a ser cada fragmento, se seleccionan al azar entre todos los creados.

2. **Almacenar variables del prefabricado:** El módulo terreno almacena las variables referentes al prefabricado que usará cuando haga sus cálculos.

3. **Modificación del terreno y decorados:** Si el módulo Terreno, tras hacer los cálculos referentes al terreno, determina que puede alojar el prefabricado, recalcula su fragmento de terreno para adaptarlo a la estructura. Si determina que no puede alojar el prefabricado se lo comunica al módulo Entorno que selecciona otro.

El módulo Entorno asigna cada prefabricado a un módulo Terreno distinto y lo quita de la lista de módulos disponibles para alojar. También se queda a la espera, disponible para realojar un prefabricado, por si algún módulo Terreno no realiza sus cálculos hasta pasado un tiempo.

5.2.5.1.1 Actores

Los actores intervinientes a lo largo de las iteraciones serán los mismos que en la primera iteración (véase sección 5.2.1.1.1).

5.2.5.1.2 Casos de uso

No se añaden nuevos casos de uso al sistema durante esta iteración. El añadido de parámetros entra dentro del caso de uso detallado en la tabla 5.1.

5.2.5.2 Diseño

El diseño realizado durante la quinta iteración redefine la clase *PoissonDiscSampling*, que pasa a ser una estructura que implementa la interfaz *IJob*, y añade parámetros y métodos a las

clases *MapGenerator* y *TerrainGenerator* para la implementación de los edificios prefabricados. Este diseño puede verse en la figura A.15.

Los parámetros y métodos que se añaden a la clase *MapGenerator* se detallan a continuación:

1. **PrebuildedTerrains:** Lista de *GameObjects* donde cada uno representa una estructura prefabricada.

2. **PrebuildedTerrainsBorderLen:** Lista de longitudes del borde de cada estructura prefabricada. Debe tener el mismo tamaño que la lista anterior.

3. **AssignPrebuilds:** Método que selecciona un *TerrainGenerator* aleatorio por cada estructura prefabricada y le pasa los parámetros necesarios para que modifique su terreno.

4. **GenerateGrid:** Método que genera una matriz de ocupación para que no se añadan elementos a la muestra de decorados en el área que ocupará la estructura prefabricada.

5. **ChangePrebuildPosition:** Método que es usado por un *TerrainGenerator* si determina que su terreno no reúne las características necesarias para alojar la estructura prefabricada.

Los parámetros y métodos que se añaden a la clase *TerrainGenerator* se detallan a continuación:

1. **normalizeBorders:** Booleano que el usuario evaluará a *true*, antes de la ejecución, si desea que se aplanen los bordes del fragmento de terreno circundantes a la estructura prefabricada.

2. **havePrebuild:** Booleano que se evalúa a *true* si un prefabricado es asignado a este fragmento de terreno.

3. **prebuildOcupied:** Matriz de ocupación del prefabricado. Será la muestra origen que se le pasará a la distribución en disco de Poisson con la posición del prefabricado marcada como ocupada, lo que no permitirá que ningún elemento dentro de ese área pase a la muestra final.

4. **prebuildIdentifier:** Identificador del prefabricado. Se usa si el fragmento de terreno no es válido para que la clase *MapGenerator* lo identifique y reasigne.

5. **BorderLen:** Longitud del borde del prefabricado.

6. **PrebuildedTerrain:** El *GameObject* que representa el prefabricado y se instancia si el terreno es adecuado.

7. **mapGenerator:** Referencia a la clase *MapGenerator* necesaria, si el fragmento de terreno no es válido, para llamar a la función *ChangePrebuildPosition*. Esta referencia solo la obtienen las clases *TerrainGenerator* seleccionadas para alojar un prefabricado.

8. **CreateShapeFromExistentMesh:** Método que modifica el terreno eliminando los polígonos de la zona ocupada por el prefabricado. Si el booleano *normalizeBorders* es evaluado a *true* modifica la altura de los polígonos restantes para adecuarlos a la altura media del fragmento de terreno.

9. **PoissonDistribution:** Método puente llamado por el método *GeneratePoissonDist*, que genera un *job* de tipo *PoissonDiscSampling* por cada muestreo necesario.

10. **GetMeshRotation:** Método que calcula las pendientes de las dos diagonales del fragmento de terreno.

11. **SetPrebuild:** Método llamado por la clase *MapGenerator* para asignar todas las variables referentes a la estructura prefabricada.

La clase *PoissonDiscSampling* recibe un rediseño total. Ya no usará la funcionalidad original de los muestreos en disco de Poisson, cuyo experimento consistía en comprobar la distancia de cada elemento con todos los demás. El nuevo funcionamiento tendrá un enfoque de causa y efecto, donde la causa será añadir un elemento a la muestra y el efecto es el bloqueo del área circundante.

Para tal propósito se definen una serie de variables detalladas a continuación:

1. **radius:** Radio de los elementos actuales. Determina el área de ocupación.

2. **xsize, ysize:** Anchura y altura de la matriz.

3. **sample:** Lista de elementos que pertenecen a la muestra, almacena el resultado de la ejecución.

4. **elementList:** Lista de elementos iniciales, que se someterán al experimento, para determinar si pertenecen a la muestra.

5. **previousSample:** Matriz de ocupación, representada mediante un array por eficiencia, que almacena todos los puntos ocupados. Es una variable de entrada y salida. Tiene 1 donde hay huecos libres y -1 donde hay huecos ocupados. Si no se ha usado todos sus valores son 0.

El funcionamiento implementado en el método heredado *Execute* se resume en 5 pasos:

- Se comprueba el primer valor de *previousSample*, si es igual a 0 el array solo ha sido creado y no guarda información previa, tras lo cual, se igualan todos los valores a 1, es decir, huecos libres. Si contiene algún valor distinto de 0 quiere decir que ya ha sido usado y contiene información previa, por lo que no se realiza ninguna acción.
- Se itera a través de todos los elementos candidatos alojados en *elementList*.
- Mediante los dos primeros valores de cada elemento, que contienen las coordenadas o posiciones dentro del fragmento de terreno, se calcula la posición que tendría que ocupar en la matriz y se comprueba si este valor es igual a 1.
- Si no es igual a 1 quiere decir que esa posición está ocupada y el elemento queda descartado.
- Si la posición es igual a 1 está libre, se iguala a -1 y se recorre la matriz en forma de rombo, cuyo centro es la posición actual, igualando todos los valores a -1 y quedando descartados para iteraciones siguientes. Se elige este método ya que es el más simple y eficiente que simula un radio de ocupación en una matriz. Tras marcar como ocupadas todas las posiciones, se añade el elemento a la muestra final.

Tras la ejecución se obtiene una lista de elementos listos para generar y una matriz de ocupación que sirve como entrada para la siguiente ejecución, garantizando que dos elementos no se solapen independientemente del tipo que sean, ya que para cada tipo de elemento decorativo se realiza una ejecución diferente.

5.2.5.3 Implementación

Para la implementación de esta iteración se modifica totalmente el *script* que contiene la clase *PoissonDiscSampling*. Se modifican los *scripts MapGenerator* y *TerrainGenerator* para añadir las nuevas características y funcionalidad. Por último, se crea una estructura prefabricada para la realización de las pruebas.

5.2.5.4 Pruebas

Todas las pruebas se realizan con misma semilla y el mismo equipo informático (véase tabla 5.4). Se mantendrá siempre el mismo rango de altura ([0, 2000]), mismos perímetros de cálculo y visualización y tamaño de entorno de 5 km^2. Se toman siempre los mismos parámetros referentes a biomas y ecosistemas de la iteración anterior, detallados en la figura 5.22, ya que no se hacen cambios referentes a este aspecto durante la iteración actual y genera un terreno muy abrupto que escenifica el peor escenario posible.

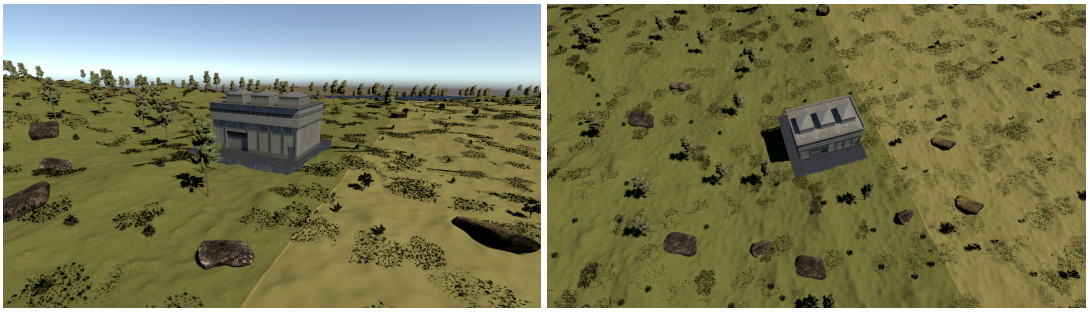
Se han realizado un total de 4 pruebas, detalladas en la tabla 5.14.

Cuadro 5.14: Pruebas realizadas en la cuarta iteración

Nº	Descripción	Resultado esperado	Resultado obtenido
1	Inclusión de un edificio prefabricado con base de 64 metros sobre terreno abrupto e irregular.	Fragmento de terreno con corte central de 64x64 metros, ajustado a los límites del prefabricado.	Fragmento de terreno con corte central de 64x64 metros, ajustado a los límites del prefabricado. (Ver figura 5.29)
2	Inclusión de un edificio prefabricado con base de 64 metros sobre terreno abrupto e irregular.	Fragmento de terreno con corte central de 64x64 metros, ajustado a los límites del prefabricado.	Fragmento de terreno con corte central de 64x64 metros, ajustado a los límites del prefabricado. (Ver figura 5.30)
3	Inclusión de un edificio prefabricado con base de 128 metros sobre terreno abrupto e irregular.	Fragmento de terreno con corte central de 128x128 metros, ajustado a los límites del prefabricado.	Fragmento de terreno con corte central de 128x128 metros, ajustado a los límites del prefabricado. (Ver figura 5.31)
4	Inclusión de un edificio prefabricado con base de 64 metros sobre terreno abrupto e irregular. Se marca la opción de normalización del terreno.	Fragmento de terreno con corte central de 64x64 metros, ajustado a los límites del prefabricado, cuyos bordes están normalizados.	Fragmento de terreno con corte central de 64x64 metros, ajustado a los límites del prefabricado, cuyos bordes están normalizados. (Ver figura 5.32)
5	Inclusión de un edificio prefabricado con base de 64 metros sobre terreno abrupto e irregular. Se marca la opción de normalización del terreno.	Fragmento de terreno con corte central de 64x64 metros, ajustado a los límites del prefabricado, cuyos bordes están normalizados.	Fragmento de terreno con corte central de 64x64 metros, ajustado a los límites del prefabricado, cuyos bordes están normalizados. (Ver figura 5.33)

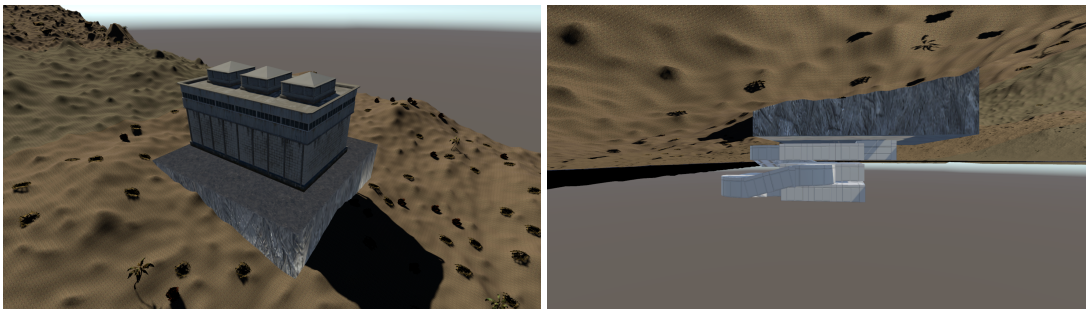
La primera y segunda prueba determinan que, con unos buenos parámetros de aceptación por parte del fragmento del terreno, el sistema recorta la parte justa y necesaria del polígono para que el prefabricado se ajuste a los límites. Esto habilita las construcciones bajo tierra ya que no existe terreno en el interior de los cimientos del prefabricado.

Sin embargo, en la tercera prueba se observa como una pendiente inadecuada deja al descubierto el recorte del terreno. Unos parámetros más restrictivos, en cuanto a la pendiente máxima que debe tener el terreno para aceptar el prefabricado, solucionan este problema, pero



(a) Vista detallada. (b) Vista aérea.

Figura 5.29: Resultados de la primera prueba



(a) Vista detallada. (b) Vista subterránea.

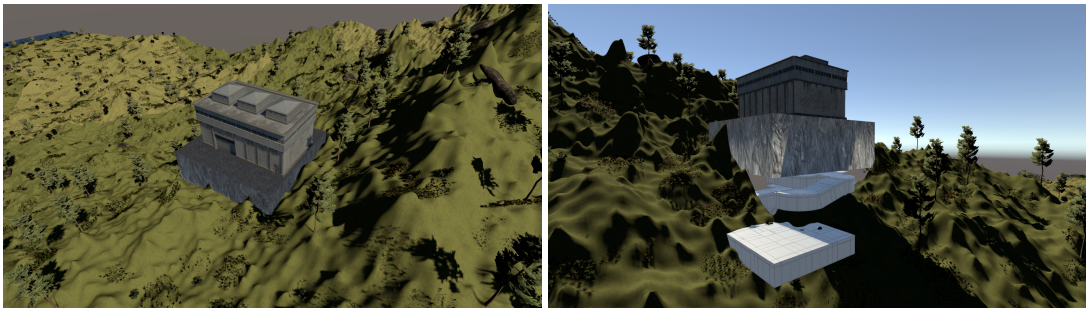
Figura 5.30: Resultados de la segunda prueba

podría llevar a que los prefabricados no se pudiesen asignar a ningún fragmento en un terreno sumamente escarpado. Por este motivo se desarrolló el sistema de normalización de bordes, que ajusta los vértices del polígono al borde del prefabricado y normaliza los valores entre estos bordes y los bordes del fragmento de terreno.

En la cuarta y quinta prueba vemos como el sistema de normalización genera antinaturalidad en el terreno. Evita la posibilidad de aperturas en el terreno independientemente de la pendiente, como se observa en la figura 5.34, pero la desnaturalización es demasiado elevada para que sea un método viable.

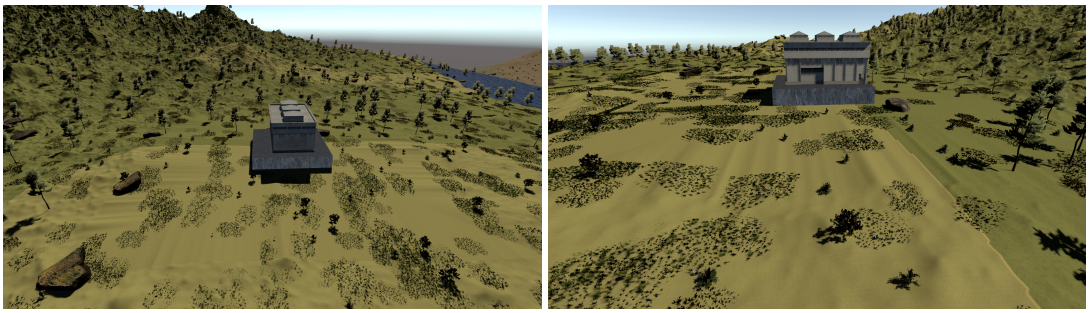
Se repiten las mismas pruebas de tiempo que en la iteración 2 para comprobar el aumento de rendimiento de la nueva distribución en disco de Poisson. Las pruebas de tiempos concluyen que se ha reducido un 20% el tiempo inicial de ejecución y se cumple el objetivo inicial de generar 1 km^2/s.

El resultado de la iteración en general es satisfactorio, la nueva distribución en disco de Poisson mejora el rendimiento un 20% y se aumenta el realismo al añadir edificios al terreno. Por otra parte, la normalización del terreno no alcanza el nivel esperado y se tendrá que modificar en futuros desarrollos.



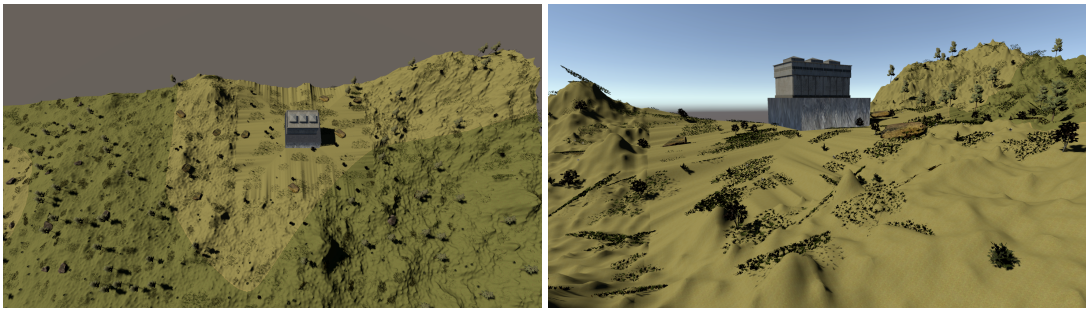
(a) Vista aérea. (b) Vista detallada.

Figura 5.31: Resultados de la tercera prueba



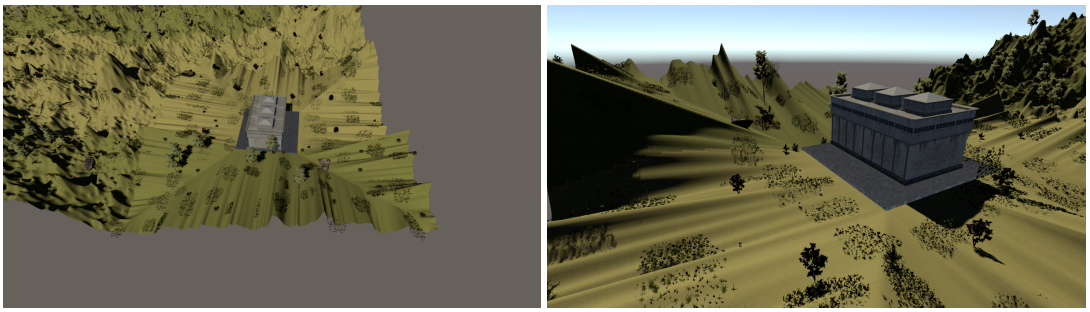
(a) Vista aérea. (b) Vista detallada.

Figura 5.32: Resultados de la cuarta prueba



(a) Vista aérea. (b) Vista detallada.

Figura 5.33: Resultados de la quinta prueba



(a) Vista aérea. (b) Vista detallada.

Figura 5.34: Resultado de normalización sobre pendiente elevada

Prueba	Radio perímetro cálculo	Radio perímetro visualización	Tiempo inicial
1	1500 metros	1000 metros	8 segundos
2	1500 metros	1000 metros	10 segundos
3	1500 metros	1000 metros	10 segundos
4	2000 metros	1500 metros	12 segundos
5	2000 metros	1500 metros	12 segundos
6	3500 metros	3000 metros	23 segundos
7	2000 metros	1500 metros	12 segundos
8	3500 metros	3000 metros	17 segundos
9	2000 metros	1500 metros	14 segundos

Cuadro 5.15: Tiempo de cálculo inicial.

Capítulo 6

Conclusiones

En este capítulo se comentan las características finales de la herramienta, si ha cumplido los objetivos, los resultados del seguimiento y costes finales.

6.1 Características y objetivos

Basándonos en los requerimientos descritos en la sección 5.1 y los objetivos que se encuentran detallados en la sección 1.2, todos los elementos se han completado satisfactoriamente. Finalmente el proyecto consiste en una herramienta que genera entornos naturales realistas de grandes dimensiones. Capaz de crear y colocar fragmentos de terreno modelados mediante mapas de ruido, de forma continua y realista, y de distribuir todo el entorno en diferentes biomas y estos en diferentes ecosistemas. Al mismo tiempo genera y coloca decorados en el terreno, respetando las leyes físicas, que son representativos del ecosistema en el que están situados.

La herramienta cumple con todos los objetivos propuestos y está especialmente pensada para desarrollos ágiles, ya que en muy poco tiempo ofrece muchos resultados personalizados. Así pues, puede funcionar de base para una multitud de sistemas basados en entornos 3D.

El principal problema durante el desarrollo fue la eficiencia, debido principalmente a que Unity basa su ejecución en un único hilo. Identificar en qué puntos se podía ejecutar el código en paralelo y adecuarlo al sistema *Jobs* de C# fueron las soluciones adoptadas.

En lo personal, se destaca el hecho de ser el primer proyecto íntegramente desarrollado por el alumno, aplicando los conocimientos adquiridos durante su formación en la Facultad de Informática y los adquiridos con investigación y desarrollo sobre las tecnologías que ofrece Unity.

Teniendo en cuenta todo lo anterior, se considera que el desarrollo ha sido satisfactorio, resultando en una herramienta versátil, con un nivel de personalización y detalle alto, facilidad de uso y totalmente automática. Esta herramienta puede ser usada como base de un video-

Total salarios	8.151,68 €
Total materiales e intangibles	107,52 €
Total proyecto	**8.259,20 €**

Cuadro 6.1: Coste total proyecto

juego o simulador, ahorrando grandes cantidades de tiempo y dinero a pequeñas y medianas empresas que deseen introducirse en este mercado.

6.2 Seguimiento y coste final

Durante la realización del presente proyecto hubo una serie de desviaciones en los tiempos de varias iteraciones, detallados a continuación:

- **Primera iteración:** El tiempo estimado inicial de esta iteración fue de 35 días. Debido al proceso acelerado de aprendizaje gracias a la enorme comunidad de Unity, se consiguió reducir en 3 días el desarrollo.
- **Tercera Iteración:** El tiempo estimado inicial de esta iteración fue de 15 días. Debido a la gran complejidad del sistema *ECS* de Unity y la depuración de errores en *multithreading*, se aumentó 2 días el tiempo de desarrollo y 1 día el tiempo de pruebas.
- **Quinta iteración:** El tiempo estimado inicial de esta iteración fue de 7 días. Debido a la implementación del normalizado de bordes y su posterior descarte, se incrementó 1 día el tiempo de desarrollo.

Debido a estas desviaciones, se modificó el diagrama de Gantt final como se puede observar en la figura 6.1. También se modificó la tabla de costes totales como se puede observar en la tabla 6.1.

6.3 Posibles aplicaciones

Esta herramienta puede ser empleadas en una gran variedad de aplicaciones que requieran un entorno natural 3D, como por ejemplo:

- Todo tipo de videojuegos que requieran un entorno natural y realista que pueda variar en cada nueva partida, multiplicando las posibilidades y tiempo de juego.
- En cualquier aplicación educativa que quiera simular ecosistemas y su flora.
- Simuladores de vehículos o aeronaves, en especial vehículos todoterreno y aviones comerciales.

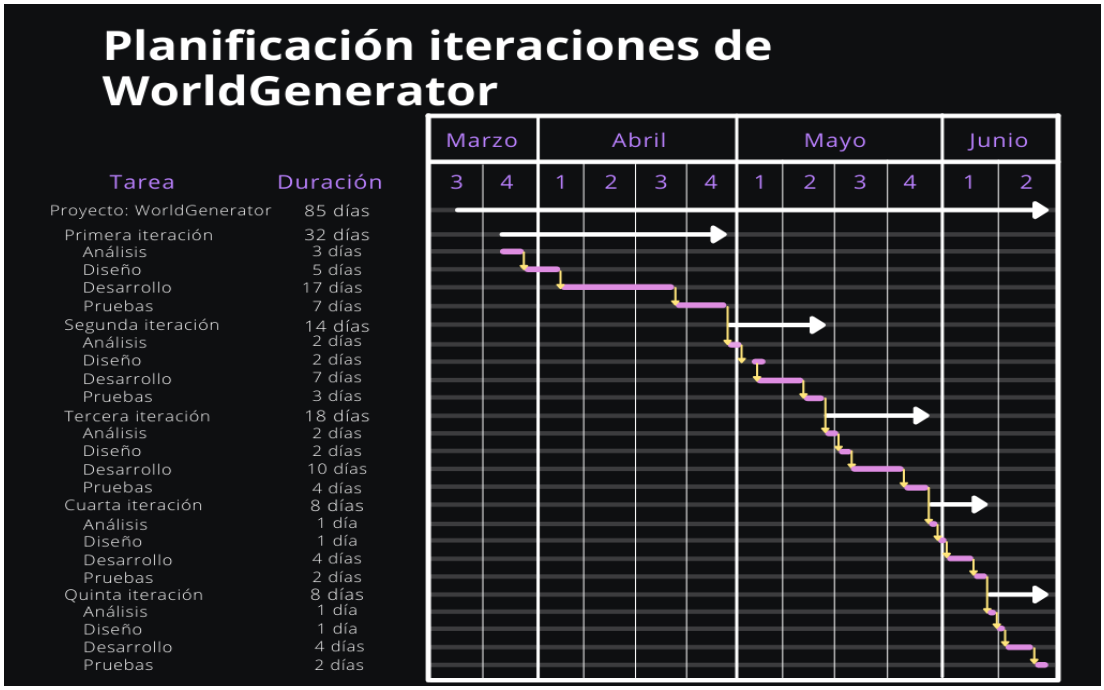


Figura 6.1: Diagrama de Gantt final

- Sistemas de realidad virtual que requieran entornos naturales realistas.

Capítulo 7

Futuros Desarrollos

La herramienta desarrollada en el presente proyecto es un producto finalizado y podría pasar a una fase de comercialización, pero al mismo tiempo se podrían añadir nuevas características que eleven el nivel de detalle, personalización y realismo.

7.1 Mejoras de la herramienta

En esta sección se detallan las posibles líneas de desarrollo futuro:

- **Implementar lista de modificadores de frecuencias y amplitudes:** El sistema actual de apilación de mapas de ruido consiste en, multiplicar la frecuencia por un valor y dividir la amplitud por otro en cada iteración. Estos valores son los mismos independientemente del número de mapas que se deben apilar. Dichos valores se podrían cambiar por una lista de pares, cada elemento sería usado en un nivel de detalle distinto, así cada mapa de ruido apilado tendría su propio modificador de frecuencia y amplitud.
- **Carreteras:** La inclusión de múltiples estructuras prefabricadas puede llevar a verlas aisladas, colocadas al azar sin naturalidad. Para solucionar esta visión se implementaría un sistema que conectaría las estructuras con carreteras. Estas se adaptarían al terreno y lo modificarían para eliminar pendientes elevadas y hoyos.
- **Ríos:** La inclusión de agua a un cierto nivel aumenta el realismo, pero sin ríos que alimenten el mar se pierde naturalidad. Para aumentar el realismo y la naturalidad se podrían trazar líneas curvas aleatorias desde los puntos más altos del terreno hasta el nivel del mar, generando posteriormente los ríos a lo largo de estas curvas.
- **Mejora del *shader* del terreno:** El *shader* que pone texturas al terreno funciona de manera muy sencilla, asignando texturas en base a ecosistemas. Para mejorar el realismo del terreno se podrían implementar combinaciones de texturas basadas en ruido, pendiente, altura u orientación.

Al mismo tiempo hay características que se consideran mejorables:

- **Colocación de decorados en terrenos muy abruptos:** Cuando el terreno es muy abrupto la colocación no es muy eficiente. Cuando un decorado coincide en un punto elevado aislado se realizan una serie de comprobaciones y cálculos que normalmente terminan con la destrucción del decorado, debido a que no existe manera de posicionarlo sin que vulnere las leyes físicas.
- **Limitaciones en la colocación de estructuras:** La colocación de estructuras prefabricadas es limitada, debido al resultado antinatural del terreno normalizado se descarta este proceso, con lo que solo un fragmento de terreno con pendiente inferior a 15 grados puede albergar un prefabricado, ya que es la única manera de garantizar que no habrá hoyos en el terreno.

Apéndices

Apéndice A

Diagramas

En este capítulo se recogen los diagramas UML realizados durante las fases de análisis y diseño a lo largo de las iteraciones.

A.1 Flujo

A.1.1 Iteración 1

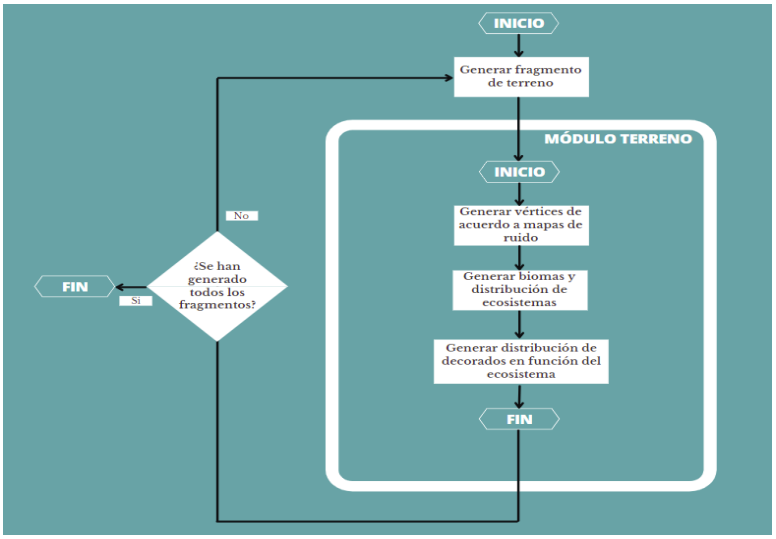


Figura A.1: Diagrama de flujo de los algoritmos de creación de entorno y terreno.

A.1.2 Iteración 2

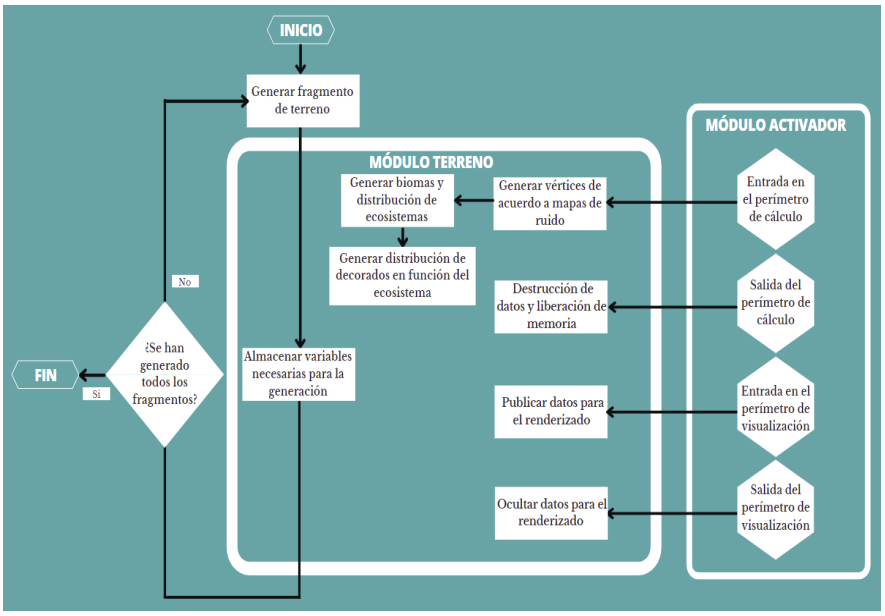


Figura A.2: Diagrama de flujo de la segunda iteración.

A.1.3 Iteración 3

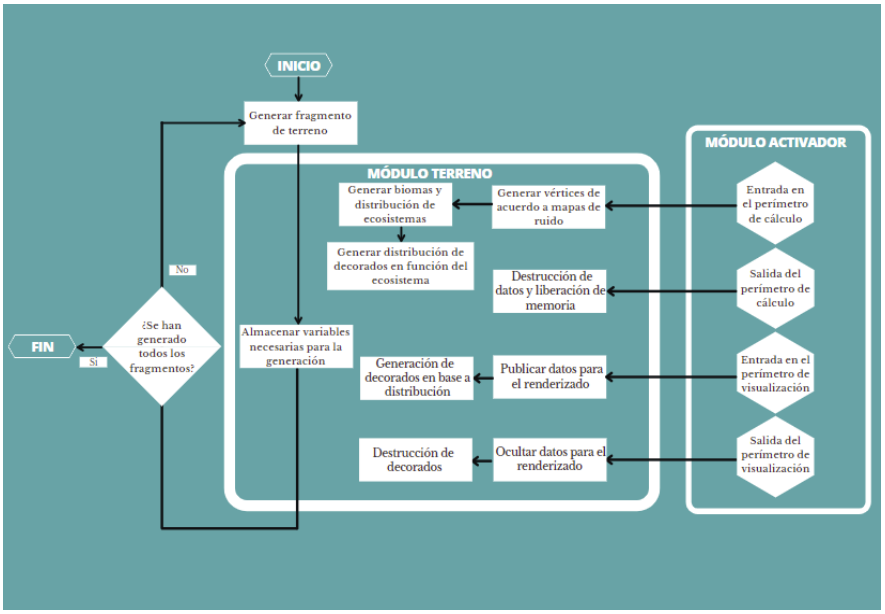


Figura A.3: Diagrama de flujo de la tercera iteración.

A.1.4 Iteración 5

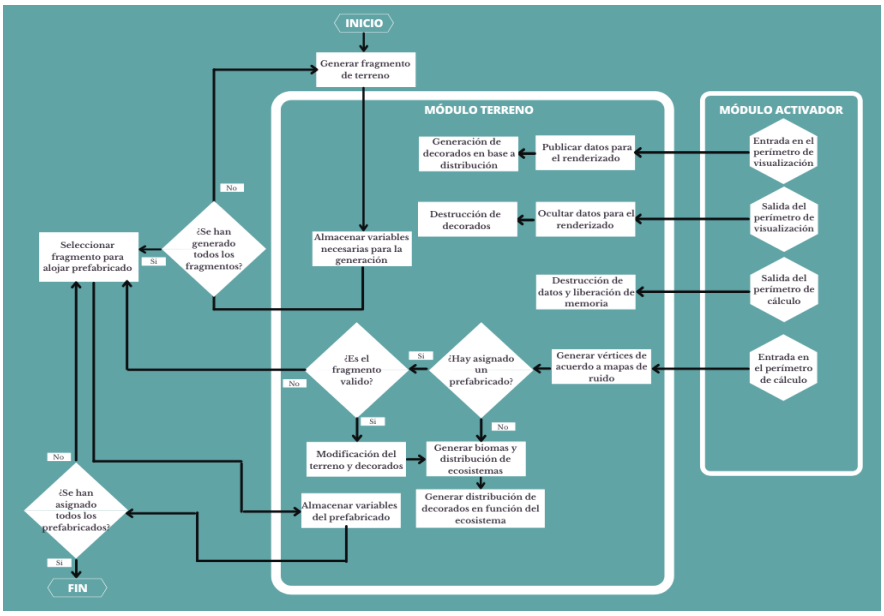


Figura A.4: Diagrama de flujo de la quinta iteración.

A.2 Casos de uso

A.2.1 Iteración 1

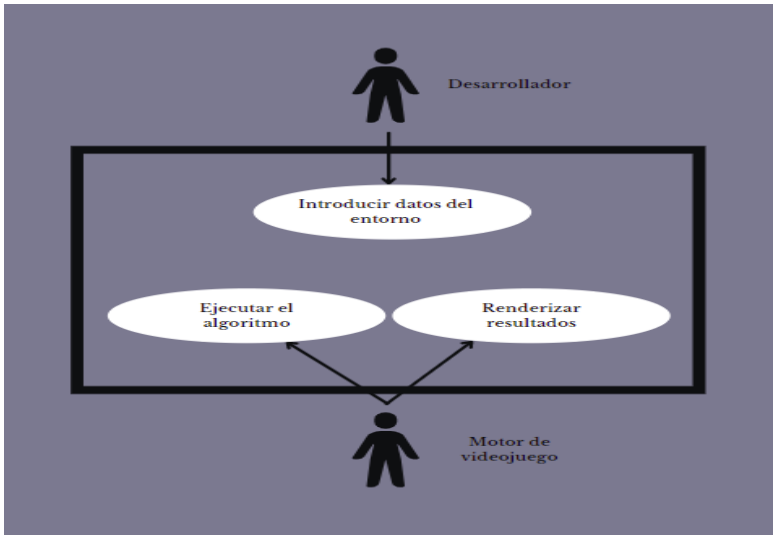


Figura A.5: Diagrama de casos de uso de la primera iteración.

A.2.2 Iteración 2

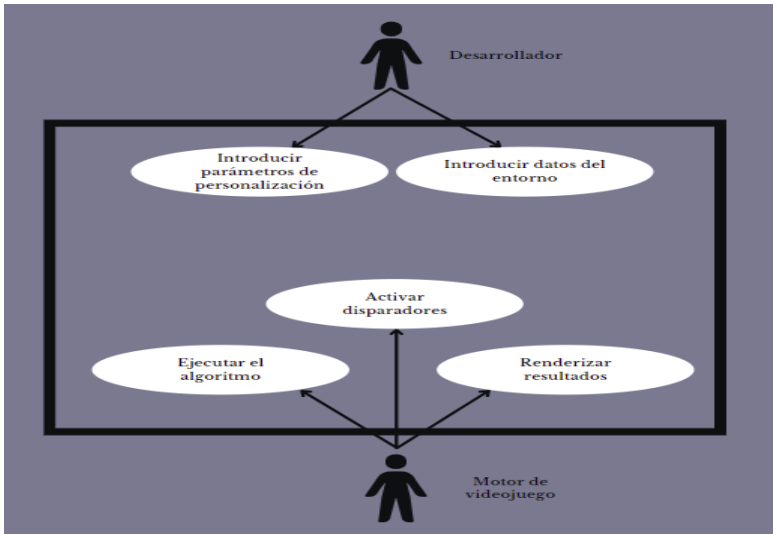


Figura A.6: Diagrama de casos de uso de la segunda iteración.

A.2.3 Iteración 3

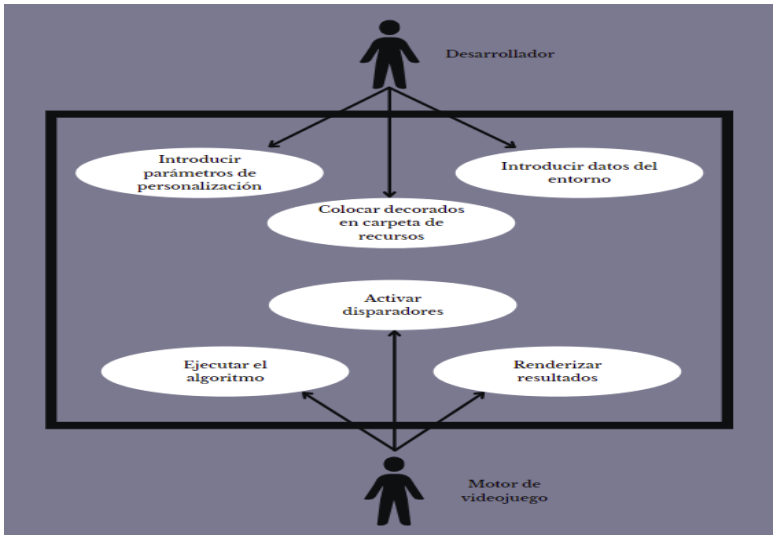


Figura A.7: Diagrama de casos de uso de la tercera iteración.

A.2.4 Iteración 4

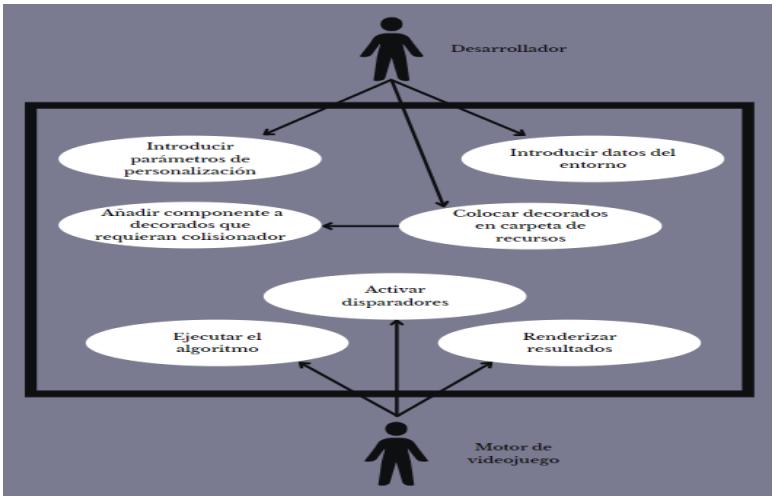


Figura A.8: Diagrama de casos de uso en la cuarta iteración.

A.3 Clases

A.3.1 Iteración 1

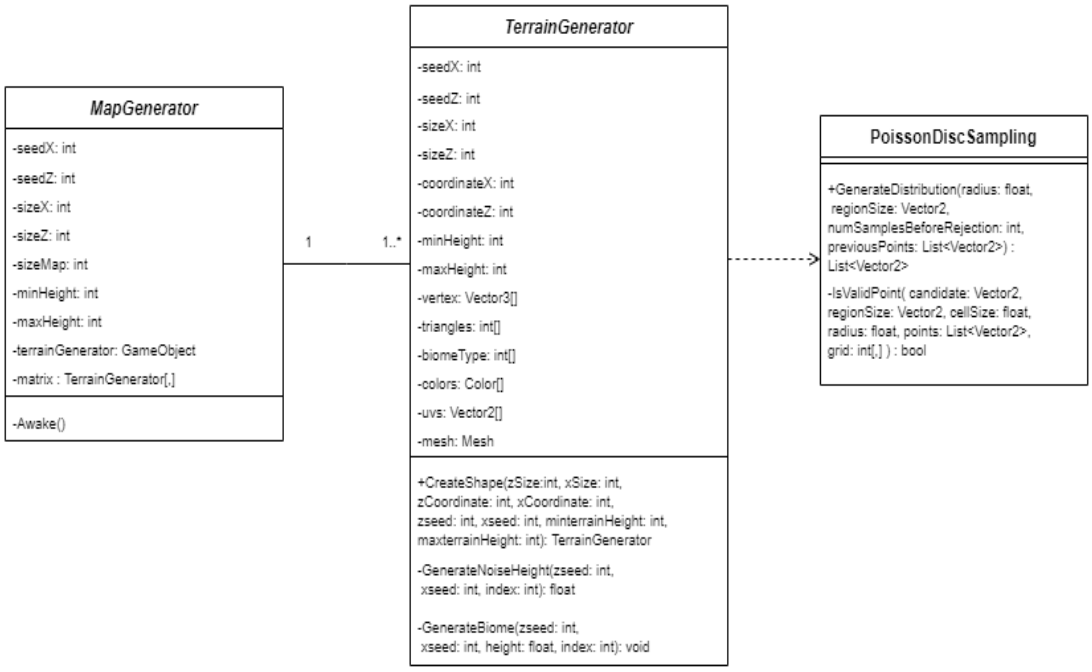


Figura A.9: Diagrama de clases de la primera iteración.

A.3.2 Iteración 2

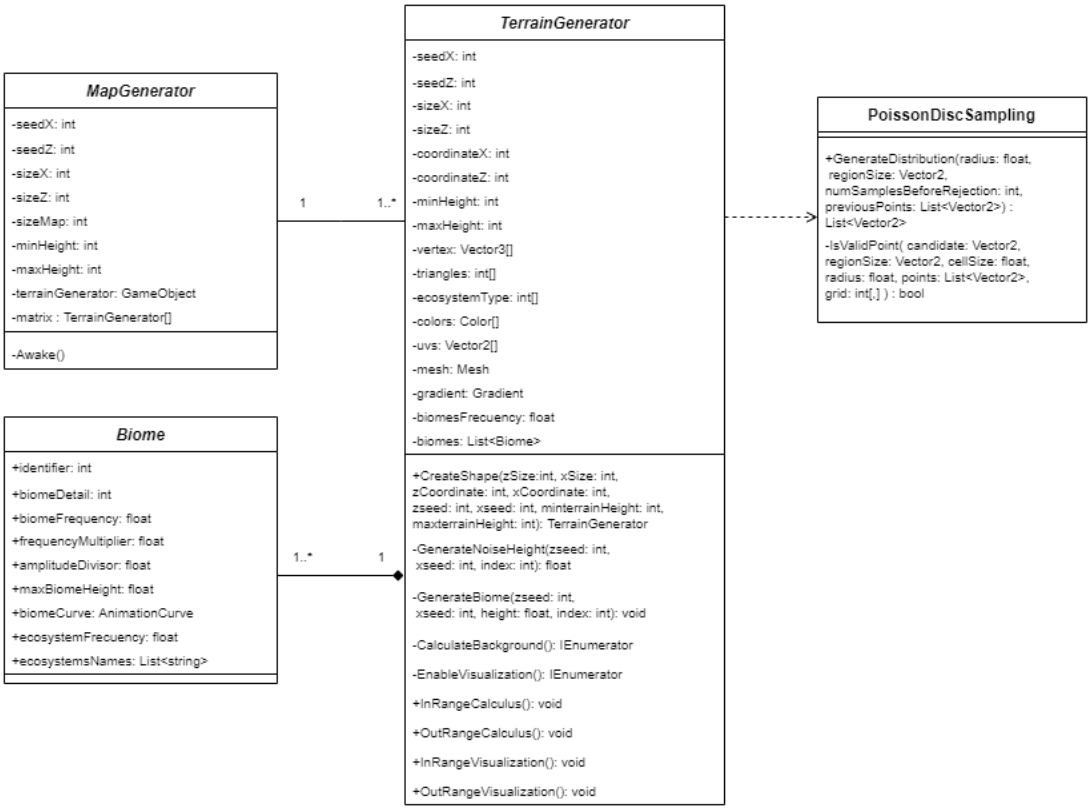


Figura A.10: Diagrama de clases de la segunda iteración, donde se añade la clase Biome.

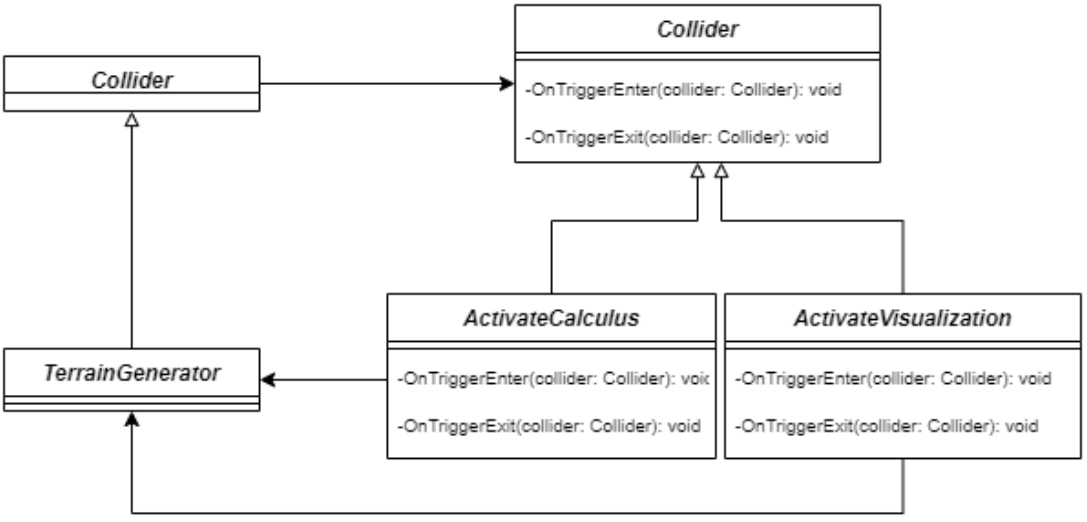


Figura A.11: Diagrama de clases correspondiente al modulo Activador.

A.3.3 Iteración 3

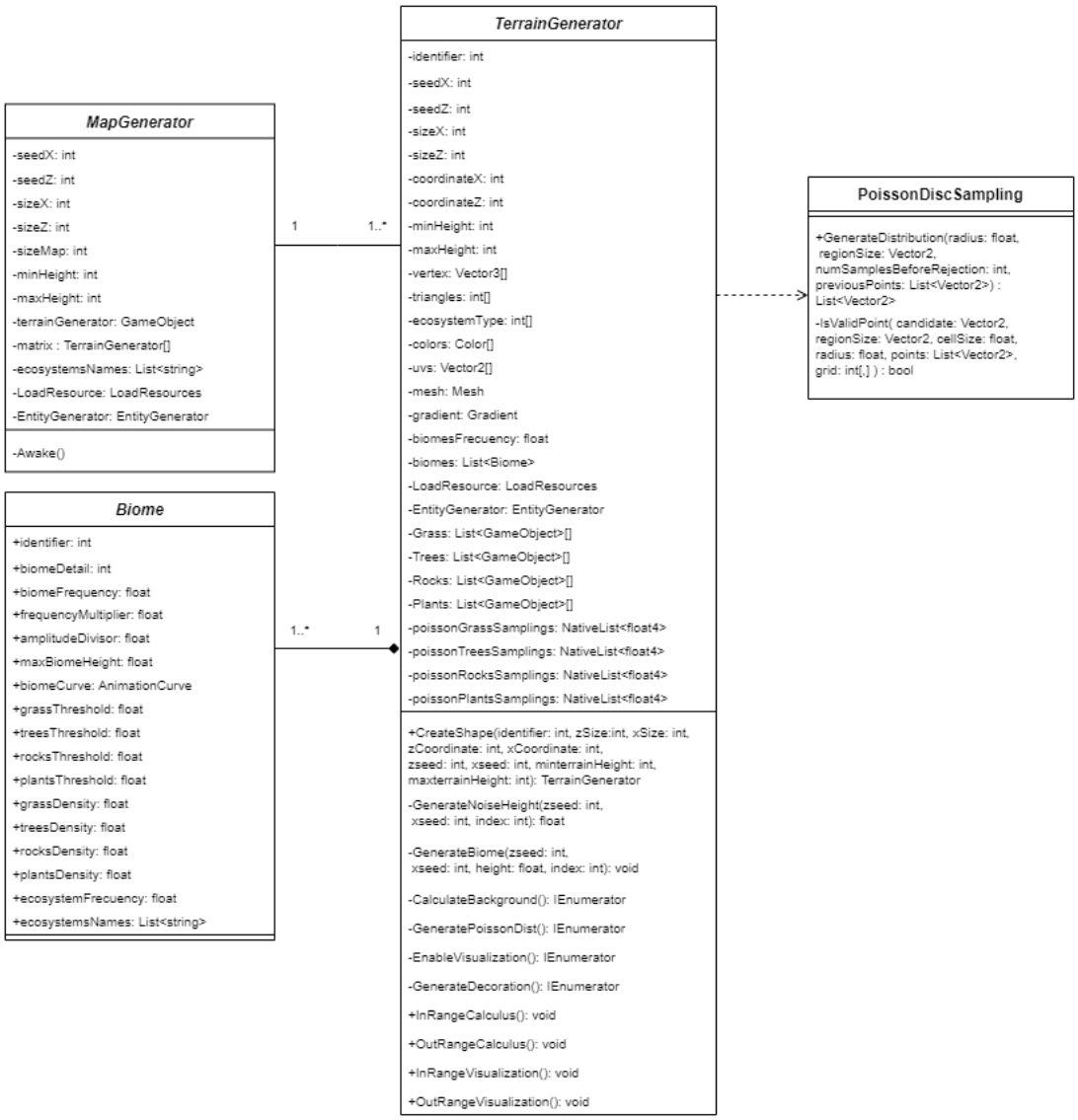


Figura A.12: Diagrama de clases de la tercera iteración, donde se añaden las características necesarias para la generación de decorados.

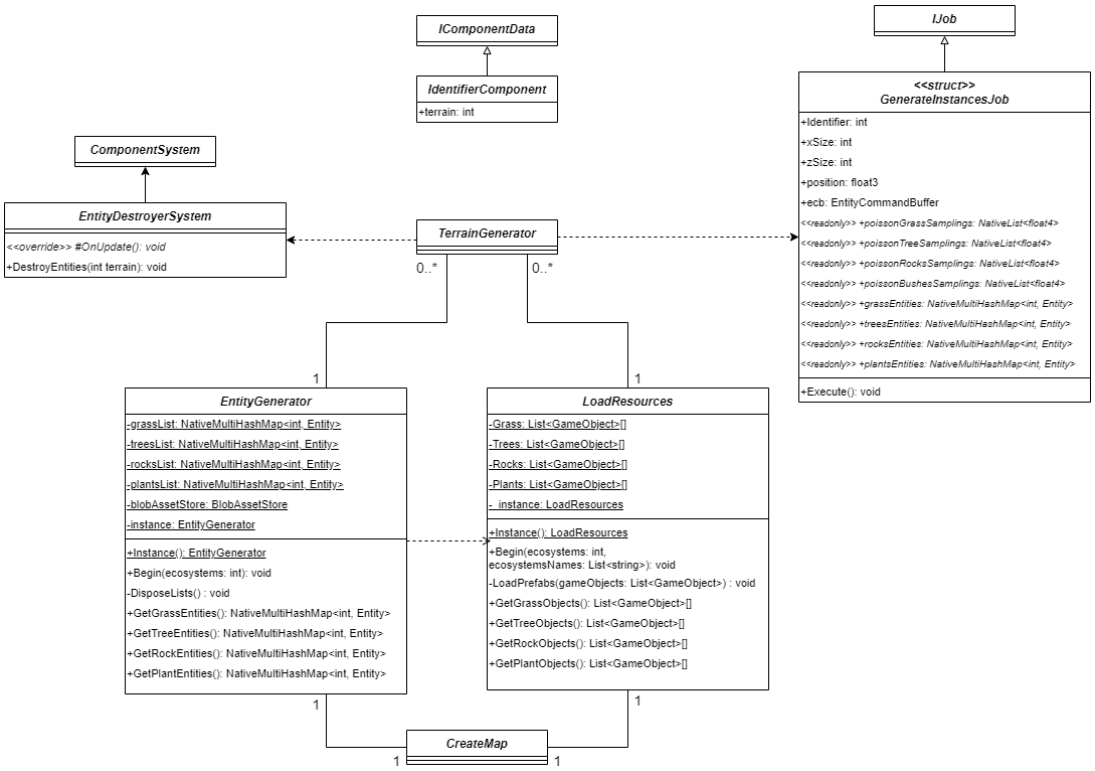


Figura A.13: Diagrama de clases de la tercera iteración, donde se observa el funcionamiento de la generación y destrucción de decorados.

A.3.4 Iteración 4

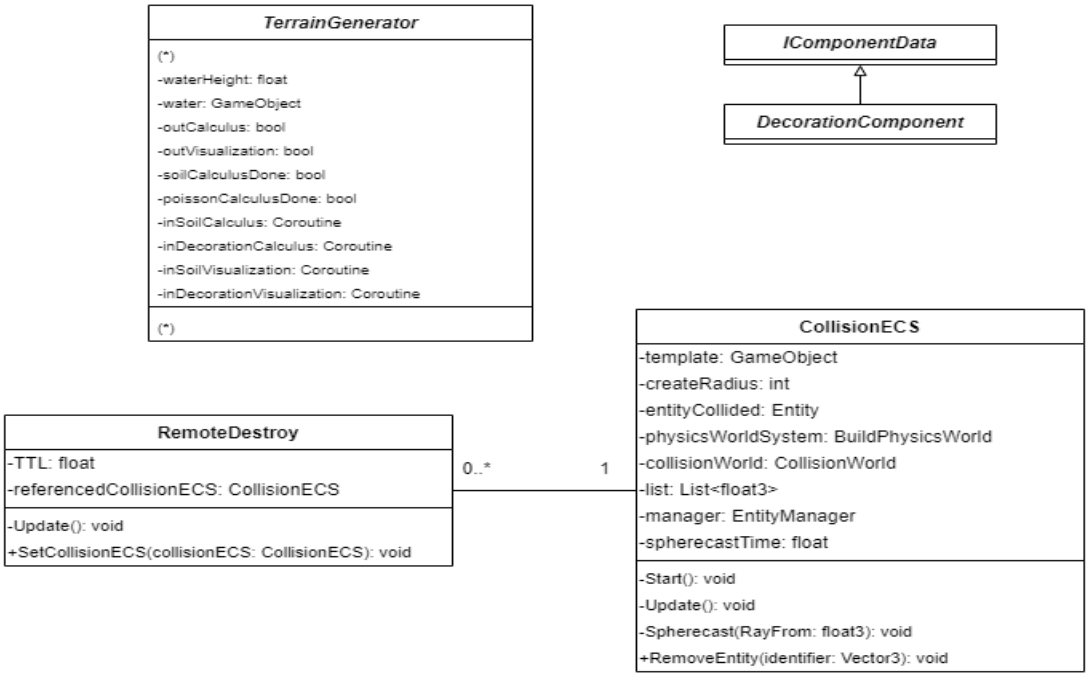


Figura A.14: Diagrama de clases de la tercera iteración, donde se añaden las características necesarias para la generación de decorados.

A.3.5 Iteración 5

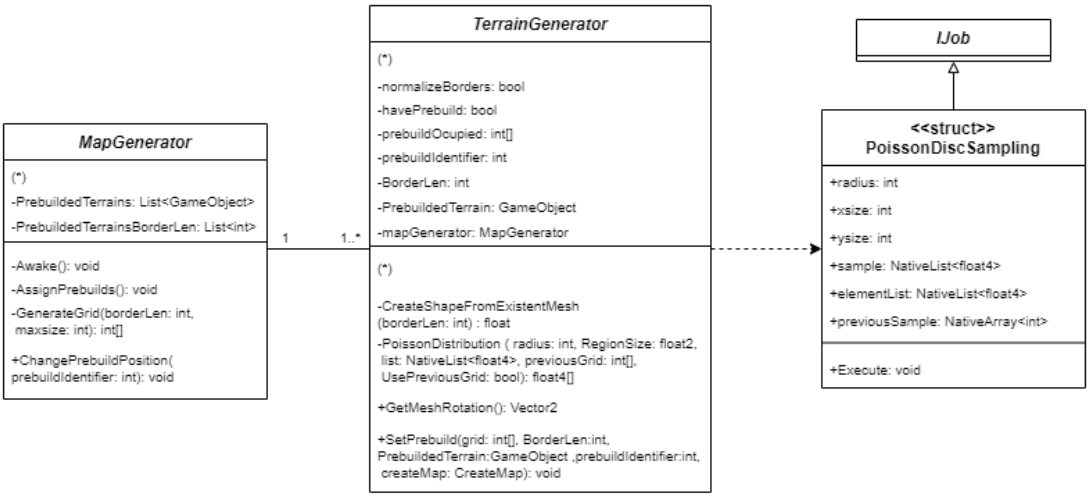


Figura A.15: Diagrama de clases de la quinta iteración.

Apéndice B

Manual de usuario

En este capítulo se detallan los manuales de instalación y uso de la herramienta.

B.1 Manual de instalación

B.1.1 Requisitos mínimos

La herramienta desarrollada durante este proyecto puede ser instalada en cualquier equipo con soporte para la versión 2019.3.x de Unity o posteriores. Se recomienda al menos 16 GB de memoria RAM para la simulación de entornos muy grandes (Superiores a 100 km^2). Se recomienda el uso de una tarjeta gráfica Nvidia GTX 1060 o equivalente o superior para el uso de entornos con gran densidad de decorados.

B.1.2 Instalación

Para la instalación se pueden seguir dos procesos sencillos detallados a continuación:

1. **Copia directa:** Consiste en copiar la carpeta *WorldGenerator* en el directorio del proyecto Unity.

2. **Importar paquete personalizado:** Para importar un paquete personalizado se deberá hacer click derecho sobre la carpeta *Assets* y seleccionar *Import package* > *Custom Package* como se ve en la figura B.1, tras lo cual se seleccionará el paquete en la carpeta correspondiente y se seleccionarán los archivos que se deseen importar como se observa en la figura B.2 (para un correcto funcionamiento se recomienda la selección de todos los componentes).

Tras la importación por parte de Unity, se deberán descargar los paquetes de los que dependen los *scripts* de la herramienta listados a continuación:

- *Entities*

- *Burst*
- *Collections*
- *Mathematics*
- *Performance Testing API*
- *Serialization*
- *Mono Cecil*
- *Jobs*
- *Scriptable Build Pipeline*
- *Platforms*

Para la instalación de paquetes se usará el *Package Manager* de Unity como se ve en la figura B.3. Si se decide empezar por la instalación del paquete *Entities*, este se encargará de descargar e importar los demás ya que tiene dependencias con todos ellos.

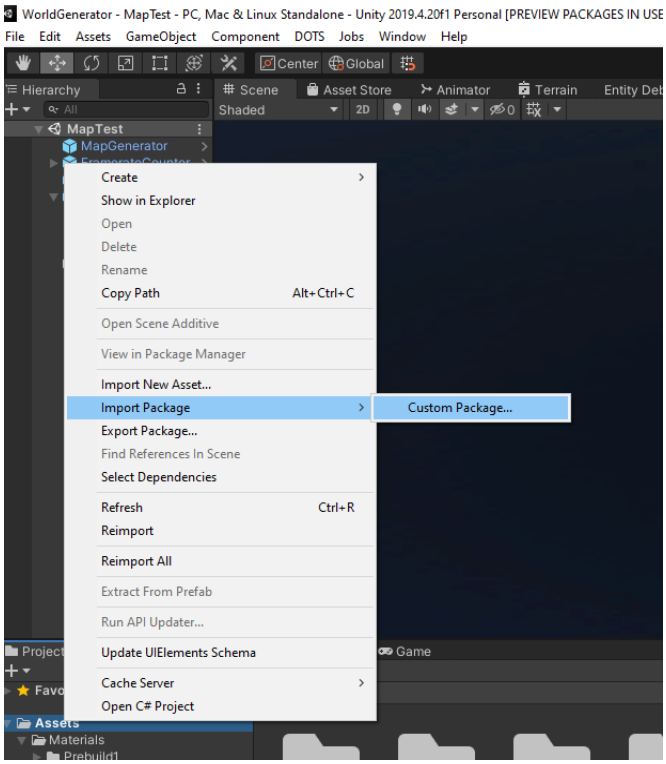


Figura B.1: Importacion de un paquete personalizado en Unity.

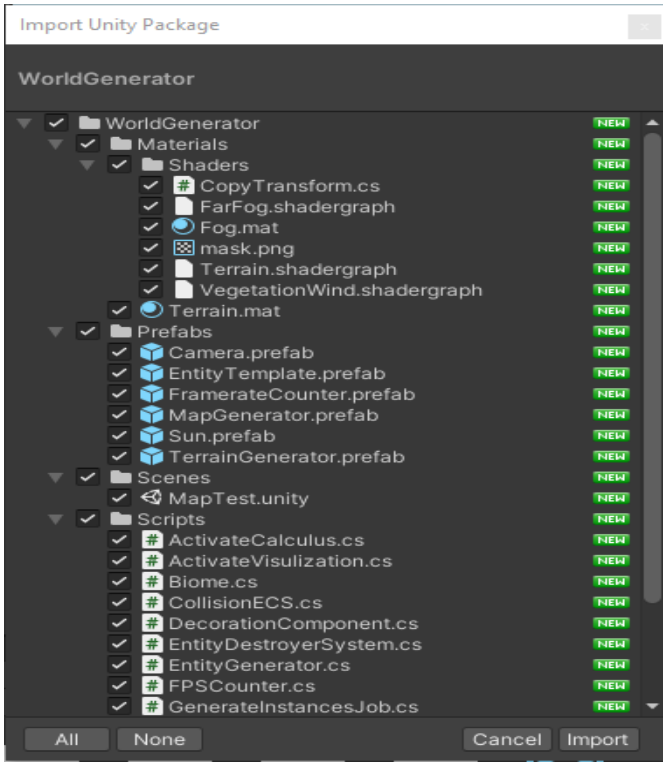


Figura B.2: Importacion de WorldGenerator en Unity.

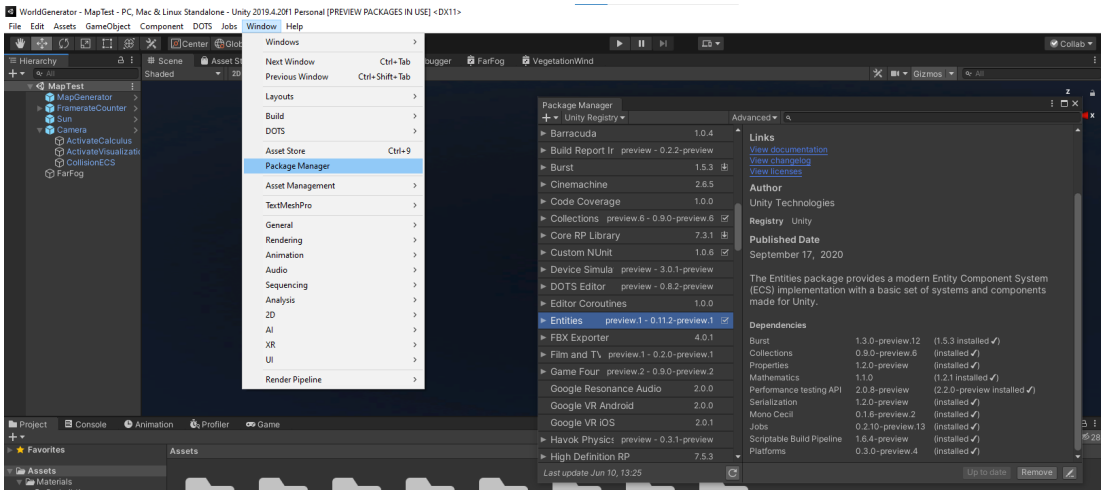


Figura B.3: Gestor de paquetes de Unity.

B.2 Manual de uso

Una vez la herramienta es importada en el proyecto de Unity, solo se necesitan dos *prefabs* en la escena a ejecutar, *MapGenerator* y *Camera*, para que se genere todo el entorno.

La configuración de los perímetros de cálculo y visualización están sujetos al radio de los *SphereColliders* de los *GameObjects* hijos de la cámara (ver figura B.4) . Se recomienda tener un perímetro de cálculo un 20% superior al de visualización.

Para la configuración de la generación de *Colliders* se debe modificar el radio del *script CollisionECS*, que también es hijo de la cámara como se observa en la figura B.5.

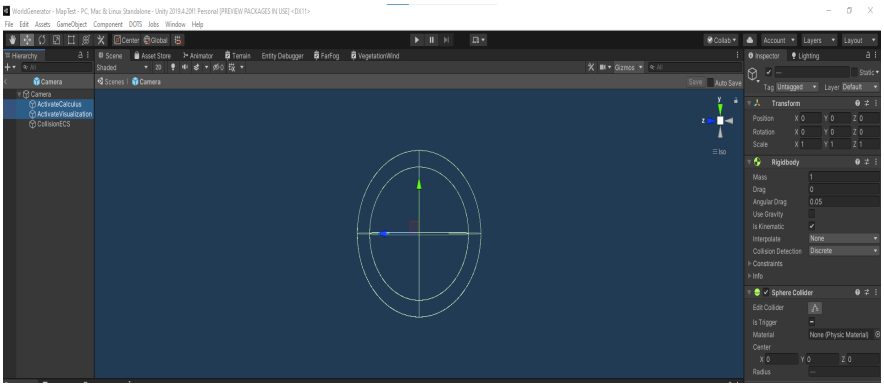


Figura B.4: Perímetros de visualización y cálculo dentro de la cámara.

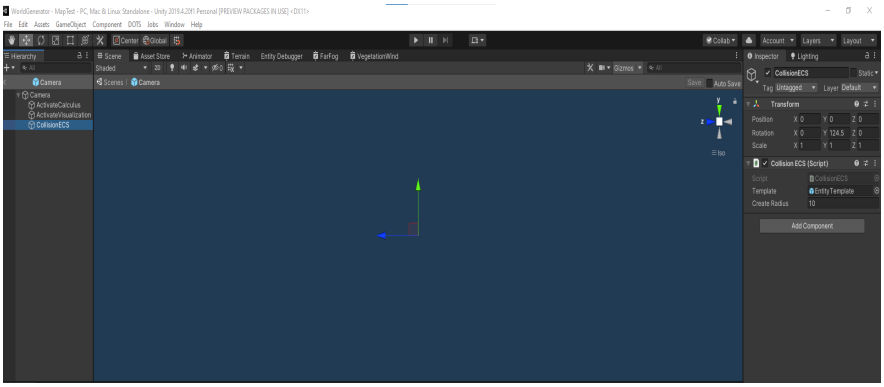


Figura B.5: *GameObject* encargado de generar los *Colliders* de las *Entities*.

Para la configuración del tamaño y semilla del entorno se deben modificar los parámetros del *script MapGenerator* detallados a continuación:

- **sizeMap:** Número de fragmentos de terreno que se crearán por fila y columna. El entorno estará compuesto por *sizeMap x sizeMap* fragmentos de terreno.
- **XSeed:** Valor de la semilla que tomará la primera variable del mapa de ruido.
- **ZSeed:** Valor de la semilla que tomara la segunda variable del mapa de ruido.
- **XSize:** Valor de la anchura de los fragmentos de terreno a generar.

- **ZSize:** Valor de la profundidad de los fragmentos de terreno a generar.
- **MinTerrainHeight:** Altura mínima del terreno.
- **MaxTerrainHeight:** Altura máxima del terreno.

Para la inclusión de estructuras prefabricadas en el terreno se deben asignar a la lista *PrebuildedTerrains*. También se deben asignar los valores de longitud de los bordes de estas estructuras a la lista *PrebuildedTerrainsBorderLen*. Estas dos listas deben contener el mismo número de elementos. Todos los parámetros de *MapGenerator* pueden observarse en la figura B.6.

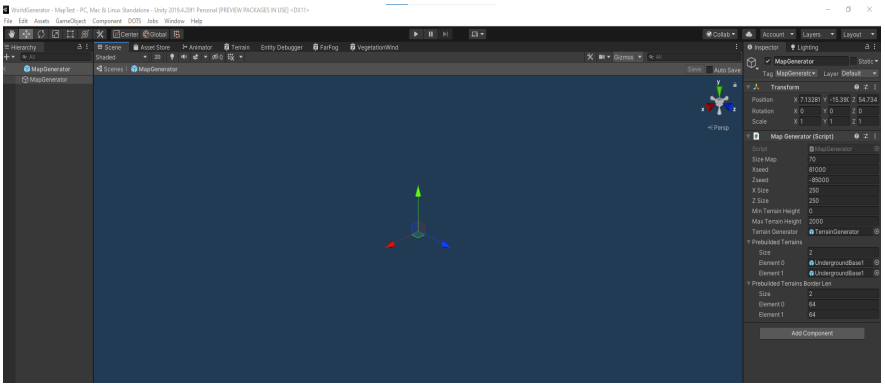


Figura B.6: Parámetros de *MapGenerator*.

Por último se puede configurar todo lo referente a biomas, ecosistemas, orogenia, vegetación y agua dentro del *script TerrainGenerator* del *GameObject TerrainGenerator*. Dichas características se pueden ver en la figura B.7 y se detallan a continuación:

1. **Gradient:** Gradiente de colores cuyo número de intervalos será el mismo que el número de ecosistemas.
2. **WaterHeight:** Nivel sobre la altura mínima del terreno en el que se generara agua.
3. **Water:** *GameObject* de tamaño 1X1 que represente un fragmento de agua.
4. **BiomesFrequency:** Frecuencia del mapa inicial de distribución de biomas. A mayor valor los biomas estarán más juntos y tendrán menos extensión, a menor valor más separados y de mayor extensión.
5. **Biomes:**Lista de biomas que compondrán el entorno, cada bioma tendrá sus características diferencias que se detallan a continuación:

- **Identifier:** Identificador del bioma, necesario para diferenciar un bioma de otro y para acceder a las características de cada uno en concreto. Cada bioma necesita tener un identificador distinto.
- **biomeDetail:** Número de mapas de ruido que se apilan para obtener el resultado final. A mayor número mayor realismo.
- **biomeFrequency:** Frecuencia inicial de repetición del bioma, número de veces que el valor del mapa de ruido oscila entre el máximo y el mínimo. Un valor muy grande generará pendientes muy pronunciadas, un valor muy pequeño generará pendientes muy pequeñas.
- **frequencyMultiplier:** Multiplicador que se aplica, en cada ciclo de detalle, a la frecuencia del bioma, para disminuir la distancia entre máximos y mínimos en el mapa de ruido de la iteración.
- **maxBiomeHeight:** Porcentaje de la altura máxima del entorno que representa la altura máxima del bioma. Es la amplitud inicial del bioma, que define el máximo y el mínimo del mapa de ruido.
- **amplitudeDivisor:** Divisor que se aplica, en cada ciclo de detalle, a la amplitud del bioma, para disminuir la magnitud de los maximos y minimos en el mapa de ruido de la iteracion.
- **biomeCurve:** Curva paramétrica que representa el contorno de las formas geológicas que aparecerán en el terreno. Es un multiplicador extra que se aplica al mapa de ruido para forzar formas concretas de terrenos.
- **ecosystemFrequency:** Frecuencia de repetición de los ecosistemas que componen el bioma. A mayor frecuencia ecosistemas más pequeños y a menor frecuencia ecosistemas más grandes.
- **ecosystemsNames:** Lista con los nombres de los ecosistemas que componen el bioma. Se usa para conocer el número de ecosistemas y se usará para acceder a los decorados de cada ecosistema.

El *GameObject TerrainGenerator* puede ser duplicado multitud de veces con configuraciones muy distintas dado que se usará como plantilla. Para hacer uso del que se requiera simplemente se habrá de pasar como parámetro al *script MapGenerator* del *GameObject MapGenerator* que se encuentra en la escena.

Una vez se tienen todos los parámetros configurados solo será necesaria la ejecución del motor de videojuego, para lo cual solo se tendrá que hacer click en el botón resaltado en la figura B.8.

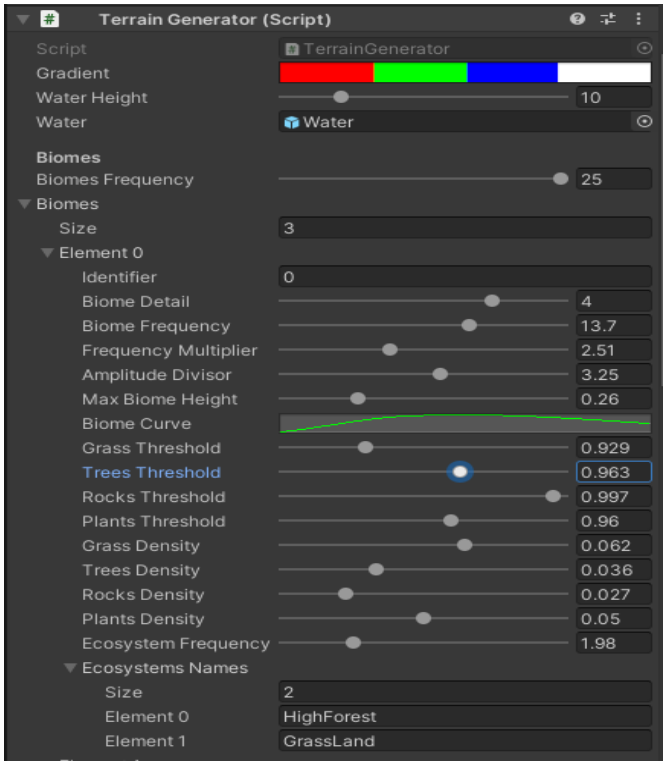


Figura B.7: Parámetros de configuración del terreno.

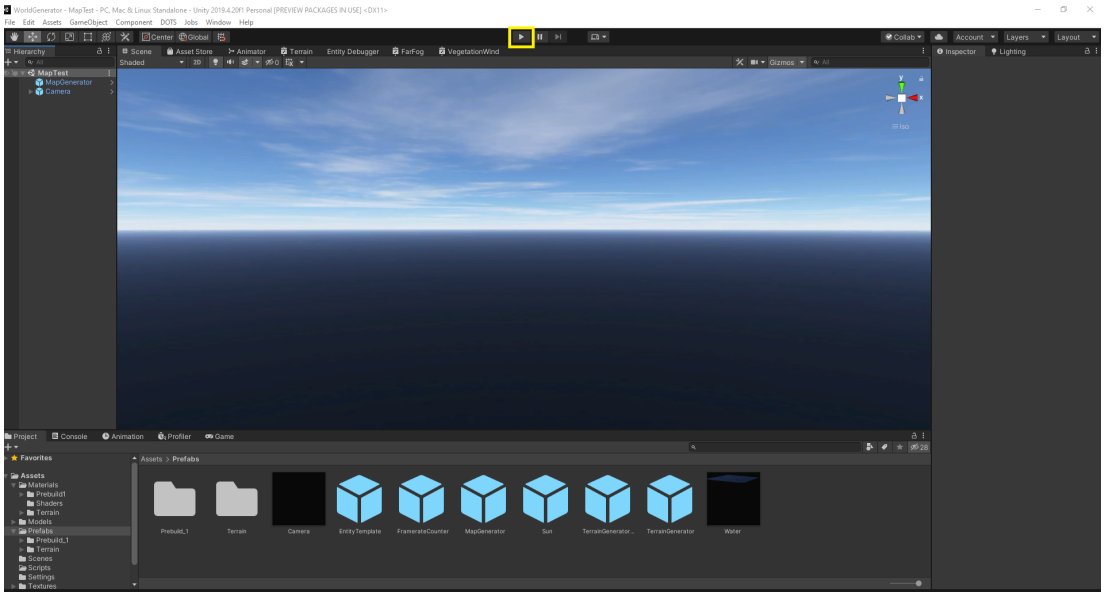


Figura B.8: Interfaz de Unity con el botón de ejecución resaltado.

Lista de acrónimos

3D *3 Dimensiones.*

ECS *Entity Component System.*

UML *Unified Modeling Language.*

AMD *Advanced Micro Devices.*

RAM *Random Access Memory.*

GHz *Giga Hercio.*

MHz *Mega Hercio.*

DDR *Double Data Rate.*

GDDR *Graphics Double Data Rate.*

KM *kilómetro.*

TTL *Time To Live.*

Glosario

Entity Component System Tecnología orientada a datos y multihilo de Unity, para más detalles ver sección 3.2.

GameObject Objeto base de Unity, contenedor de los componentes que definen un elemento, para más detalles ver sección 3.2.

Mesh Componente de Unity que representa cualquier modelo 3D, para más detalles ver sección 3.2.

MeshRenderer Componente de Unity que se encarga de renderizar un modelo 3D con un material, para más detalles ver sección 3.2.

MonoBehaviour Clase que ofrece Unity para definir comportamientos, para más detalles ver sección 3.2.

Script Documento escrito en un lenguaje de programación que contiene una funcionalidad.

Prefab *GameObject* plantilla, creado antes de la ejecución, que sirve como referencia para instanciar copias.

Bibliografía

[1] "Página oficial de gaia," 2021. [En línea]. Disponible en: https://www.procedural-worlds.com/products/professional/gaia-pro/

[2] "Página oficial de terrain composer," 2021. [En línea]. Disponible en: http://www.terraincomposer.com/

[3] "Página oficial de world creator," 2021. [En línea]. Disponible en: https://www.world-creator.com/

[4] "Página oficial de minecraft," 2021. [En línea]. Disponible en: https://www.minecraft.net/

[5] "Página oficial de no man´s sky," 2021. [En línea]. Disponible en: https://www.nomanssky.com/

[6] "Página oficial de rust," 2021. [En línea]. Disponible en: https://rust.facepunch.com/

[7] "Página oficial de unreal engine," 2021. [En línea]. Disponible en: https://www.unrealengine.com/

[8] "Página oficial de unity," 2021. [En línea]. Disponible en: https://unity.com/

[9] "Entity component system manual," 2021. [En línea]. Disponible en: https://docs.unity3d.com/Packages/com.unity.entities@0.17/manual/index.html

[10] K. Perlin, "Improving noise," 2021. [En línea]. Disponible en: https://mrl.cs.nyu.edu/~perlin/paper445.pdf

[11] "Documentación de unity," 2021. [En línea]. Disponible en: https://docs.unity3d.com/ScriptReference/Mathf.PerlinNoise.html

[12] C. Yuksel, "Sample elimination for generating poisson disk sample sets," 2021. [En línea]. Disponible en: https://core.ac.uk/download/pdf/189678027.pdf

[13] B. B., *A Spiral Model of Software Development and Enhancement.* IEEE, 1988.

[14] R. S. Pressman, *Software Engineering. A practitioner's Approach.* McGraw-Hill Education, 2001.

[15] "Página oficial de canva," 2021. [En línea]. Disponible en: https://www.canva.com/es_es/graficos/

[16] "Salario medio de un analista programador," 2021. [En línea]. Disponible en: https://es.indeed.com/career/analista-programador/salaries?from=top_sb

[17] "Salario medio de un programador informático junior," 2021. [En línea]. Disponible en: https://es.indeed.com/career/programador-inform%C3%A1tico-junior/salaries

[18] "Salario medio de un tester," 2021. [En línea]. Disponible en: https://es.indeed.com/career/tester/salaries?from=top_sb

[19] V. Sarcar, "Facade pattern," in *Design Patterns in C#.* Springer, 2020, pp. 163–176.

[20] V. Sarcar, "Prototype pattern," in *Design Patterns in C#.* Springer, 2020, pp. 27–55.

[21] V. Sarcar, "Observer pattern," in *Design Patterns in C#.* Springer, 2020, pp. 269–286.

[22] V. Sarcar, "Singleton pattern," in *Design Patterns in C#.* Springer, 2020, pp. 3–26.